SUB

Programación de S.O. en IA-32

J. Enríquez - LU 36/08 - juanenriquez@gmail.com
N. Gleichgerrcht - LU 160/08 - nicog89@gmail.com
J. Luini - LU 106/08 - jluini@gmail.com

Resumen

En el siguiente trabajo desarrollaremos un pequeño kernel que configurará la CPU desde cero de manera tal de permitirnos correr dos tareas dadas por la cátedra, en intervalos regulares de tiempo. Dicho núcleo será capaz de, no solo correr las tareas, sino además de atender interrupciones (excepciones) y de mostrar mensajes por pantalla.

Keywords

Sistemas Operativos - Kernel.asm - Bochs - IA-32

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://exactas.uba.ar

Índice

1. Introducción

En el presente trabajo práctico relacionado con la programación de sistemas operativos para la arquitectura IA-32, nos dedicaremos a mostrar la realización y el funcionamiento paso a paso de un pequeño *kernel*. Un kernel es una pieza de software que se encarga, de alguna manera, de coordinar el uso de recursos de hardware entre las distintas aplicaciones de software disponibles. Es de cierta forma un puente entre los programas y el hardware de la pc; una especie de columna vertebral del sistema operativo.

En nuestro caso se implementó un kernel bastante simple para cumplir con los requerimientos del enunciado del trabajo práctico. En la siguiente sección desarrollaremos punto a punto el funcionamiento del mismo, aunque destacaremos ahora los pasos más importantes que este realiza:

- Paso de modo real (compatibilidad con el procesador Intel 8086) a modo protegido (Intel 80286 en adelante).
- Inicialización y configuración de la *Global Descriptor Table* (GDT). Carga de segmentos de código y datos.
- Habilitación de la unidad de paginación de memoria y configuración de la misma para cumplir con lo pedido en el ejercicio.
- Carga de las rutinas que manejan las interrupciones del procesador como así, las rutinas que manejan las excepciones del mismo.
- Configuración de las *Task State Segment* (TSS) y la subsecuente carga de las mismas a la GDT de manera de poder manejar tareas en el sistema y luego armar la rutina que las intercambia a intervalos regulares.

2. Ejercicio 1

A pesar que Bochs lo tenga habilitado, fue necesario habilitar el gate A20 para poder direccionar a toda la memoria del sistema.

Además se deshabilitaron las interrupciones del procesador mediante la instrucción `cli` para evitar que el curso de ejecución sea interrumpido de forma errónea ya que en este momento del programa las tablas de interrupciones (IDT) no han sido cargadas y por lo tanto, de llegar una interrupción el sistema se reinicia en forma no deseada.

Por otro lado utilizamos una `struct` en C para cargar los descriptores de la GDT según correspondían con lo pedido en la consigna. Recordemos que la GDT es una tabla que tiene descriptores de segmento, de tareas y de interrupciones.[1]

De esta forma, se cargaron los descriptores de segmento de memoria de la siguiente manera de acuerdo a lo requerido por Intel como así también la presencia de un descriptor de segmento nulo al inicio de la tabla.

Basado en esta definición y en lo pedido en el ejercicio se tiene la siguiente tabla de descriptores de segmento:

ID	Pos. en memoria	Descripción
1	0x0	Descriptor Nulo
2	0x8	Código
3	0x10	Dato
4	0x18	Memoria de Video

A continuación describimos los aspectos más relevantes de los descriptores anteriores.

1. Descriptor nulo. Este descriptor es requerido por Intel. La idea de este descriptor es que en caso de que la tabla no esté correctamente definida, su sola presencia al inicio, hace saltar una excepción.

2. Descriptor de segmento de código: es un segmento de nivel 0 (DPL = 0). Tiene un límite (máximo offset que se le puede sumar a la base) de 0xfffff que con la granularidad en 4 kbytes, (G = 1) dan los 4 gigabytes disponibles en la memoria del sistema. ya que la base del mismo está en la dirección 0.

3. Descriptor de segmento de datos: este segmento es casi igual al descrito anteriormente exceptuando el tipo (TYPE = 2, en vez de ser TYPE = A).

4. Descriptor del segmento de la memoria de video: este es un segmento de datos parecido al anterior exceptuando el límite y la base (que son 0x0f9f y 0xb8000 respectivamente) que permiten direccionar únicamente a la memoria de video. Además para poder lograr esto, la granularidad del mismo es a byte (G = 0). También como los anteriores tiene nivel de privilegio cero (DPL = 0).

[1] Para ver en detalle como se llenó la struct ver el archivo adjunto gdt.c

Tras deshabilitar las interrupciones, cargamos la tabla anterior mediante la instrucción `lgdt` que guarda el descriptor de la gdt en el registro `gdtr`.

Una vez concluido este paso, se procede a habilitar el bit `PE` del registro de control `CR0` mediante:

```
mov     eax, cr0
or      eax, 01h
mov     cr0, eax
```

El último paso para completar la transición a modo protegido consiste en settear el registro `CS` (Code Segment) a la posición que se le asignó al mismo en la GDT, paso se realiza mediante la ejecución de un jump far de las siguientes características: `jmp posición_en_gdt_segmento_código : punto_a_continuar`

En nuestro caso, `jmp 0x08:modoprotegido`

Una vez ya en modo protegido, se procedió a inicializar con valores correctos los demás registros de segmento. En esta oportunidad setteamos a `ES` como un puntero al segmento de la memoria de video, y el resto de los registros de forma tal de apuntar al segmento de datos.

```
;Apuntamos 'es' a la memoria de video y los demás al segmento de datos
mov ax, 0x10    ; segmento de datos
mov bx, 0x18    ; memoria de video
mov ds, ax
mov fs, ax
mov gs, ax
mov ss, ax
mov es, bx    ; es apunta a la memoria de video
```

En último lugar, para el ítem que pide dibujar un recuadro del tamaño de la pantalla, se utilizó el segmento definido que comprende a la memoria de video (ES). Se la recorrió de forma tal de dibujar un caracter (una cara feliz) en los bordes mediante el siguiente código:

```
    ; Pintamos de negro la pantalla
            mov ecx, 80*25    ;toda la pantalla
            xor esi, esi

            mov ax, 0x0000    ; negro, ningún caracter

            cleanPantalla:
                mov [es:esi], ax
                add esi, 2    ; avanza al siguiente caracter
            loop cleanPantalla

            ; Dibujamos bordes horizontales
            mov ah, 0x0F        ; blanco brillante, fondo negro
```

```
mov al, 0x02        ; caracter cara
xor esi, esi
mov edi, 80*24*2
mov ecx, 80
bordeHor:
    mov [es:esi], ax
    mov [es:edi], ax
    add esi, 2
    add edi, 2
loop bordeHor

; Dibujamos bordes verticales
xor esi, esi
mov edi, 79*2
mov ecx, 25
bordeVer:
    mov [es:esi], ax
    mov [es:edi], ax
    add esi, 80*2
    add edi, 80*2
loop bordeVer
```

3. Ejercicio 2

En este ejercicio se nos pide configurar y habilitar el modo de paginación para estos procesadores IA-32. El mismo nos permite ver a la memoria como un conjunto de páginas de tamaño fijo.

Esto se hizo de la siguiente manera. Se definieron los directorios de páginas (cada directorio contiene hasta 1024 selectores de tablas de página que a su vez pueden contener 1024 páginas propiamente dichas).

En nuestro caso, definimos dos directorios de tablas de páginas (uno por cada tarea) y además en cada directorio, una única tabla de páginas. Esto se realiza de esta forma para cumplir con el mapa de memoria solicitado.

Se empieza a ejecutar el siguiente pseudo-código[2] a partir de la dirección 0xA000 (d...)

[2]el código real de la siguiente rutina está en el archivo `paging.asm`

```
;defino directorio pintor (0xA000)
dirección_primera_tabla OR 3 ; pone los flags: read/write y present en 1
for(i = 1; I<1024; i++)
tabla_nula ; no se utilizan, se ponen en cero

;defino dir traductor (0xB000)
dirección_primera_tabla OR 3 ; pone los flags: read/write y present en 1
for(i = 1; I<1024; i++)
tabla_nula ; no se utilizan, se ponen en cero

;tabla_pintor(0xC000)
; (...) para cada valor en esta tabla cumpliendo con el mapa de memoria de la tarea
; se lo asigno a donde corresponde o se escribió cero de forma tal de no poder ser accedid
;tabla_traductor (0xD000)
; (misma observación que para la tabla anterior)
```

Vale aclarar que en los casos en el que la memoria esté mapeada se settean los primeros dos bits de forma tal de que la página en cuestión sea de tipo *read/write - present*.

Una vez cargados los directorios y las tablas de páginas correspondientes con las instrucciones...

```
mov eax, 0xA000
mov cr3, eax ; se cr3 -> dirección del dir. de página
```

...se procede a habilitar el modo de paginación mediante la rutina:

```
mov eax, cr0
or eax, 0x80000000  ; se settea el bit de paginación en cr0
mov cr0, eax
```

Luego para el inciso b) del ejercicio en cuestión es escribir en la memoria de video pero utilizando el sistema de paginación recientemente habilitado. Esto se realiza mediante la posición de memoria 0x13000 que ha sido mapeada en ambos directorios de páginas a la memoria de video. El algoritmo empleado es una típica rutina de escritura en memoria de un string, en este caso el mensaje "Orga2 SUB".

```
; 2b - Escribir "Orga2   SUB!!!" en la pantalla
mov     ecx, mensaje_len

; Usamos 0x13000 porque apunta a la memoria de video
; escribimos desde la posición (1,1) (segunda fila y columna)
mov     edi, 0x13000 + 80 * 2 + 2

 ; letras verdes, fondo azul
mov     ah, 0x1A
mov     esi, mensaje
```

```
.ciclo:
mov al, [esi]
mov [edi], ax
inc esi
add edi, 2
loop .ciclo

jmp fin_mensaje
mensaje: db "Orga 2   SUB!!!"
mensaje_len equ $ - mensaje
fin_mensaje:
```

4. Ejercicio 3

Para el ejercicio 3, se nos pide completar las entradas necesarias de la `IDT` (Interrupt Descriptor Table) de forma tal de asociar una rutina de interrupción para cada excepción del procesador y para la interrupción del reloj. Las interrupciones de excepciones muestran por pantalla cual de estas tuvo lugar, es decir, si hubo un fallo de página, por ejemplo, se imprime por pantalla el mensaje ‘‘`EXCEPTION: PAGE FAULT`’’.

Esto se logró luego de completar la IDT de forma correcta y asociando cada entrada de la IDT con el segmento de código definido anteriormente y con la rutina de atención de interrupciones que corresponde a dicha excepción.

Las rutinas de atención de excepciones están descriptas en dos archivos de `C`: `isr.c` e `isr.h`. El código de las mismas es en general similar y lo que hace es mostrar por pantalla que excepción se produjo.

```
msgisr0: db 'EXCEPCION: División por cero'
msgisr0_len equ $-msgisr0
_isr0:
        mov edx, msgisr0
        IMPRIMIR_TEXTO edx, msgisr0_len, 0x0C, 0, 0, 0x13000
        jmp $
    iret
```

La particularidad de esta rutina es que el último parámetro que indica el lugar en donde se escribe el mensaje corresponde con la dirección `0x13000` que es a donde fue mapeada (vía paginación) la memoria de video.

Especial tratamiento tuvo la rutina de atención de la interrupción del clock del microprocesador (`_isr32`) que se encarga de llamar a la función `next_clock` provista por la cátedra que dibuja un reloj en la pantalla, que con cada interrupción va cambiando su forma. Dicha rutina tiene el siguiente aspecto:

```
 ; Interrupción del reloj
```

```
_isr32:
    cli
    call next_clock
    ;(...)        Aquí se cambia de tareas (ver ej 4).
    sti
    iret
```

Para que todo lo anterior tenga sentido, es necesario agregar al archivo principal `kernel.asm` instrucciones que carguen en memoria la IDT mediante la función `idtFill()`[3]. Además, se resetean los pics 8259 de forma tal de remapear los dichos controladores a un espacio de interrupciones designado a los dispositivos de entrada-salida. De otra manera, la configuración "de-facto" entraría en conflicto porque esas direcciones están reservadas por intel (error de diseño de IBM). Entonces se settea un offset para el pic master en 0x20 y 0x28 para el pic slave (recordemos que se tienen dos pics conectados en cascada)

En último lugar, se le pasa el descriptor de la IDT (el `IDT_DESC`) y cargarlo en registro `IDTR` mediante la instrucción `lidt`.

```
; Ejercicio 3

      ; TODO: Inicializar la IDT
      call idtFill
      ; TODO: Resetear la pic
      pic_reset:
          mov al, 11h
          out 20h, al

          mov al, 20h
          out 21h, al

          mov al, 04h
          out 21h, al

          mov al, 01h
          out 21h, al

          mov al, 0xFF
          out 21h, al

          mov al, 0x11
          out 0xA1, al

          mov al, 0x28
          out 0xA1, al

          mov al, 0x02
          out 0xA1, al
```

[3]Presente en el archivo `idt.c`

```
mov al, 0x01
out 0xA1, al

mov al, 0xFF
out 0xA1, al


; TODO: Cargar el registro IDTR
lidt[IDT_DESC]
```

5. Ejercicio 4

En último lugar y respondiendo a la consigna final se nos requiere configurar el entorno para poder en primer lugar tener tareas en nuestro sistema y poder luego, hacer cambios entre ellas como lo haría un sistema operativo a gran escala.

Para ello completamos en primer lugar las `TSS` (Task State Segment) de las tareas *pintor.tsk* y *traductor.tsk*. Como sabemos, las TSS son esenciales para el manejo de tareas ya que estas contienen información esencial en lo que respecta a las mismas como ser el estado de los registros del procesador durante la ejecución del proceso en cuestión, los punteros a la pila, los registros `EFLAGS` y `EIP`, la tarea ejecutada antes que la actual y el valor del descriptor del directorio de páginas asociado a dicha tarea.

De esta manera nos armamos las TSS en memoria gracias a una `struct` provista en el archivo `tss.h` que nos permitió cargar ordenadamente los datos de cada tarea. Se las agrupó en un arreglo de nombre `tsss` y se llenó de la siguiente forma: en el primer lugar la "tarea *dummy*" (explicado más adelante), luego el pintor, y por último, la TSS correspondiente al traductor. Incluimos a modo de ejemplo ilustrativo como queda la TSS correspondiente al pintor[4]:

```
(tss) {
     (unsigned short) 0,    //ptl
     (unsigned short) 0,    //unused0
     (unsigned int) 0,    //esp0
     (unsigned short) 0,    //ss0
     (unsigned short) 0,    //unused1
     (unsigned int) 0,    //esp1
     (unsigned short) 0,    //ss1
     (unsigned short)  0,    //unused2
     (unsigned int) 0,    //esp2
     (unsigned short) 0,    //ss2
     (unsigned short) 0,    //u3
     (unsigned int) 0xA000,    //cr3 //
     (unsigned int) 0x8000,    //eip
     (unsigned int) 0x202,    //eflags
     (unsigned int) 0,    //eax
```

[4]el resto de las entradas del arreglo tsss están en el archivo tss.c

```
        (unsigned int) 0,    //ecx
        (unsigned int) 0,    //edx
        (unsigned int) 0,    //ebx
        (unsigned int) 0x15FFF,    //esp
        (unsigned int) 0x15FFF,    //ebp
        (unsigned int) 0,    //esi
        (unsigned int) 0,    //edi
        (unsigned short)  0x10,    //es
        (unsigned short)  0,
        (unsigned short)  0x8,    //cs
        (unsigned short)  0,
        (unsigned short)  0x10,    //ss
        (unsigned short)  0,
        (unsigned short)  0x10,    //ds
        (unsigned short)  0,
        (unsigned short)  0x10,    //fs
        (unsigned short)  0,
        (unsigned short)  0x10,    //gs
        (unsigned short)  0,
        (unsigned short)  0,    //ldt
        (unsigned short)  0,
        (unsigned short)  0,    //dtrap
        (unsigned short)  0xFFFF//iomap
    },
```

Es importante aclarar como se "llenaron" los campos de esta tss. En primer lugar, como podemos ver, tenemos una pila y un *stack pointer* por cada nivel de privilegio (0 a 2). Como en nuestro trabajo práctico no hay cambios de privilegios de ningún tipo, las pilas correspondientes a los anillos 1,2 se dejaron en cero puesto que no se usan. Lo mismo ocurre con la de nivel 0 pero por distintas razones, ya que en ningún momento la misma es usada, se decidió settearla en 0. Por otro lado, se pusieron los valores que corresponden a los registros que apuntan a segmentos de código y datos tal como se los configuró en ejercicios previos. Además no podemos evitar mencionar la dirección 0x15fff como el inicio de la pila de la aplicación (ver mapa de memoria) y no la 0x15000 como uno podría pensar intuitivamente (recordemos que la pila "crece para bajo"). En último término, el otro campo que llenamos que nos interesa mencionar es el del registro cr3 que se relaciona con el descriptor del directorio de páginas.

En siguiente término, para que algo de lo anteriormente dicho tenga sentido, es necesario inicializar los descriptores de las tareas en la GDT. Volvimos al archivo `gdt.c` y agregamos los descriptores correspondientes con cada tarea (incluso de la tarea *dummy*). He aquí un ejemplo:

```
/* EL DESC DE LA TAREA DEL PINTOR */
      (gdt_entry){
              (unsigned short) 0x67,  //limite [0:15]
              (unsigned short) 0,     //base [16:31]
              (unsigned char) 0x0,            //base []
              (unsigned char) 0x9,            //tipo [24]
```

```
            (unsigned char) 0,              //S [25]
            (unsigned char) 0,              //DPL [26:27]
            (unsigned char) 1,              //P [28]
            (unsigned char) 0x0,            //limite [16:19]
            (unsigned char) 0,              //AVL [20]
            (unsigned char) 0,              //L [21]
            (unsigned char) 1,              //DB [22]
            (unsigned char) 0,              //G [23]
            (unsigned char) 0x00            //BASE [24:31]
    },
```

Observemos que si bien son bastante distintos a los descriptores de segmento que teníamos anteriormente, nos gustaría recalcar la presencia del S=0 indicando que es un descriptor de sistema, y por otro lado, base y límite en 0x0 y 0x67. El segundo número es un requisito (tamaño de la tss-1), el primero se debe a que en esa dirección se debe cargar la posición en memoria del arreglo tsss que en un principio no sabemos donde está cargado. Es por eso, que setteamos esos valores dentro del archivo `kernel.asm` mediante la siguiente rutina que se encarga de recorrer dicho arreglo (cada tss tiene un tamaño fijo de 104 bytes) y de llenar con los valores que corresponden en cada descriptor de la gdt.

```
; Ejercicio 4

    ; TODO: Inicializar las TSS

    ; TODO: Inicializar correctamente los descriptores de TSS en la GDT
    xchg bx, bx
    mov eax, gdt
    add eax, 0x20

    mov ebx, tsss

    mov [eax+2], bx
    shr ebx, 16
    mov [eax+4], bl
    mov [eax+7], bh

    add eax, 8

    mov ebx, tsss
    add ebx, 104

    mov [eax+2], bx
    shr ebx, 16
    mov [eax+4], bl
    mov [eax+7], bh

    add eax, 8

    mov ebx, tsss
```

```
        add ebx, 104
        add ebx, 104

        mov [eax+2], bx
        shr ebx, 16
        mov [eax+4], bl
        mov [eax+7], bh

; en este punto se han puesto los valores correctos en la gdt
        ; TODO: Cargar el registro TR con el descriptor de la GDT de la TSS actual

        mov ax, 0x20
        ltr ax
```

Una vez corregidas las entradas de la GDT, se procede a cargar el registro `TR` *Task Register* con el valor de la tarea actual mediante la instrucciones escritas arriba.

Luego la ejecución continua habilitando los pics 8259 y las interrupciones:

```
   ; TODO: Habilitar la PIC
         pic_enable:
            mov al, 0x00
            out 0x21, al
            mov al, 0x00
            out 0xA1, al
        ; TODO: Habilitar Interrupciones
        sti
```

...y finalmente se cambia a la primera tarea con el siguiente "salto lejano" del estilo `jmp segmento:offset` donde "segmento" es en realidad la dirección en la GDT de un descriptor de TSS. Esto producirá de manera efectiva, el cambio de tarea. `jmp 0x28:0`

En este punto logramos pasar a la primer tarea que teníamos. Ahora es necesario, de alguna manera, poder intercambiar a intervalos regulares entre las tareas. pintor y traductor Para esto se modificó la rutina que dibujaba el reloj en la pantalla, incorporándole el salto entre tareas. Entonces la misma queda así:

```
 ; Interrupción del reloj
_isr32:
    cli
    call next_clock

    ;paso de tarea
    cmp BYTE [isrTarea], 0x0
    jne pasarAPintor
    pasarATraductor:
        mov BYTE [isrTarea], 0x1
```

```
    jmp 0x30:0
    sti
    iret
pasarAPintor:
    mov BYTE [isrTarea], 0x0
    jmp 0x28:0
    sti
    iret
```

Como podemos apreciar, la rutina anterior chequea el valor de una posición de memoria y según este sea 1 ó 0, ejecuta una tarea o la otra. De esta manera las tareas se van ejecutando a intervalos regulares. Ambos ramas de la ejecución culminan con un `sti` para habilitar las interrupciones y con un `iret` para volver correctamente al punto de ejecución anterior.

Como prometimos más arriba, no aclaramos que era la tarea *dummy*. Recordemos que cuando se hace un cambio de tareas se guarda el contexto de la tarea actual en su correspondiente TSS y se carga el contexto de la próxima tarea a ejecutar. En el primer paso, si no tenemos una TSS "dummy" cuando carguemos la primera tarea vamos a salvar el contexto de la tarea en curso porque así es la mecánica. Ahora, el problema es que si no tenemos una TSS para la tarea actual entonces cuando se produzca el cambio, no va a existir un lugar a donde volcar el contexto de la tarea saliente, consecuentemente se produce un error y se interrumpe la ejecución. Este observación vale para la tarea cero. Para los otros cambios no es necesario hacer más cosas que el `jmp far` que vimos antes.

6. Conclusiones

A lo largo del trabajo pudimos apreciar el funcionamiento de las distintas estructuras internas que ofrece el microprocesador a la hora de programar sistemas operativos y tuvimos la oportunidad ver como son los pasos básicos y esenciales que realiza todo sistema multitarea cosa que a nivel de usuario no es posible (y hasta parece trivial).

Estamos convencidos que un conocimiento profundo de las capas inferiores del software es necesario en pos de entender como funcionan las cosas y, consecuentemente, poder pr—ogramar en forma más eficiente.