DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Grupo Smooth criminal

Organización del Computador II

DC - UBA

Trabajo Práctico N°3

Integrante	LU	Correo electrónico
Horacio Avendaño-Lucas	129/14	horacio.avendanio.lucas@gmail.com
Brian Goldstein	27/14	brai.goldstein@gmail.com
Maximiliano Paz	251/14	m4xileon@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Ejercicio 1

El objetivo de este ejercicio es pasar de modo real a modo protegido.

Punto a)

Se define la Tabla de Descriptores Globales (GDT) con cuatro descriptores que identifican cuatro segmentos. Dos segmentos se definen para código, uno de nivel 0 y el otro de nivel 3, y dos para datos, de la misma forma. Un descriptor de la GDT presenta una estructura como se puede apreciar en la figura 1.1.

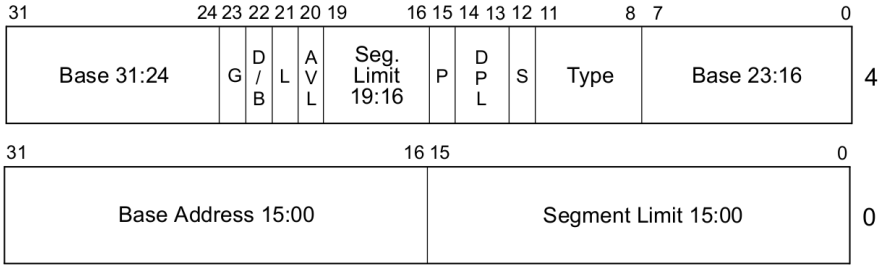

Figura 1.1: Estructura de un registro de la GDT.

De modo que, para nuestro fines, se completan esos descriptores de la siguiente manera:

Limit	0x1F3FF
Base	0x00000000
Type	0x08
S	0x01
DPL	0x00
P	0x01
AVL	0x00
L	0x00
DB	0x01
G	0x01

Cuadro 1: Segmento de código de nivel 0.

Limit	0x1F3FF
Base	0x00000000
Type	0x08
S	0x01
DPL	0x03
P	0x01
AVL	0x00
L	0x00
DB	0x01
G	0x01

Cuadro 2: Segmento de código de nivel 3.

Limit	0x1F3FF
Base	0x00000000
Type	0x02
S	0x01
DPL	0x00
P	0x01
AVL	0x00
L	0x00
DB	0x01
G	0x01

Cuadro 3: Segmento de datos de nivel 0.

Limit	0x1F3FF
Base	0x00000000
Type	0x02
S	0x01
DPL	0x03
P	0x01
AVL	0x00
L	0x00
DB	0x01
G	0x01

Cuadro 4: Segmento de datos de nivel 3.

Estos descriptores se ubican en las posiciones 8, 9, 10 y 11 de la GDT pues las primeras 7 posiciones están reservadas según el enunciado. Así, los segmentos de código de nivel 0 y 3 se ubican en las posiciones 8 y 9 respectivamente, mientras que los segmentos de datos de nivel 0 y 3, en las posiciones 10 y 11 respectivamente.

Una vez definido estos descriptores, se carga la dirección base de la GDT en el registro GDTR por medio de la instrucción LGDT.

Punto b)

Para pasar a modo protegido, se actualiza el bit 0 del registro CR0 en 1. Esto se realiza de la siguiente manera:

```
mov eax, cr0
or  eax, 1
mov cr0, eax
```

Luego, se realiza el siguiente *jump*:

```
jump 0x40:modoprotegido
```

Con este *jump*, se actualiza el contenido del selector de segmento CS haciendo referencia a la posición del segmento de código de nivel 0 en la GDT. Como la posición de ese descriptor es 8 entonces el índice del selector se configura con 0x08. Por lo tanto, se actualiza el selector de segmento CS con el valor 0x40.

Figura 1.2: Estructura de un selector de segmento.

A su vez, se actualizan los selectores de segmento DS, ES, GS y SS haciendo referencia a la posición del segmento de datos de nivel 0 en la GDT, es decir, se actualizan esos selectores de segmentos con el valor 0x50.

Por último, se configura la base de la pila del *kernel* en la dirección 0x27000 como se especifica a continuación:

```
mov ebp, 0x27000
mov esp, ebp
```

Punto c)

Se define un segmento adicional en la GDT que describe el área de pantalla en memoria que sólo puede ser utilizado por el *kernel*. La configuración de este descriptor es el siguiente:

Limit	0x00FBF
Base	0x000B8000
Type	0x02
S	0x01
DPL	0x00
P	0x01
AVL	0x00
L	0x00
DB	0x01
G	0x00

Cuadro 5: Segmento de video.

Esta área de memoria tendrá, como se puede ver, un tamaño de 4032 bytes con base en la dirección 0xB8000. Además, este descriptor se ubica en la posición 12 en la GDT. Para hacer efectivo el uso de dicho descriptor, se actualiza el selector de segmento FS haciendo referencia a la posición de éste en la GDT, es decir, se actualiza el selector de segmento FS con el valor 0x60.

Para probar el funcionamiento de la segmentación, se imprime un pixel de color rojo en la parte superior izquierda de la pantalla. El resultado obtenido es el esperado y se puede apreciar en la figura 1.3.

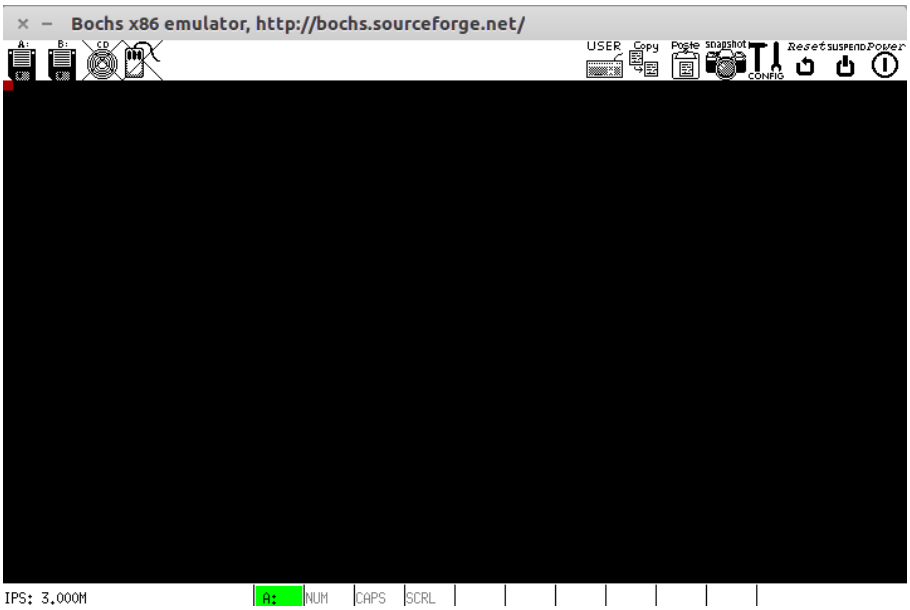

Figura 1.3: Impresión de un pixel rojo en la pantalla para probar la segmentación configurada.

Una vez comprobado el funcionamiento de la segmentación, se configura el valor del selector de segmento FS con 0x50, lo cual lo vincula con el segmento de datos de nivel 0.

Punto c)

Se inicializa la pantalla pintando el área del mapa de color gris y las barras inferiores correspondientes a los jugadores de color rojo y azul, como se puede apreciar en la figura 1.4.

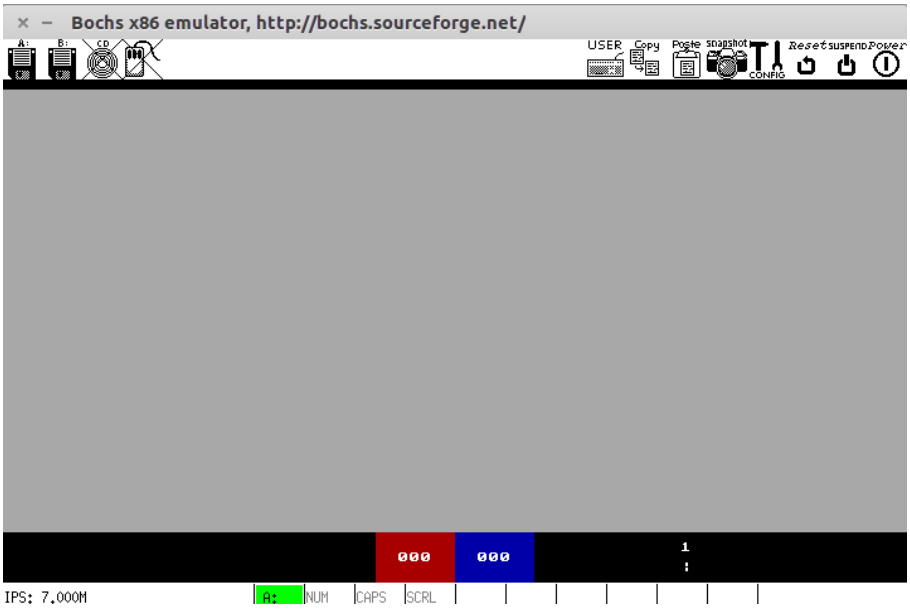

Figura 1.4: Inicialización de la pantalla.

2. Ejercicio 2

El objetivo de este ejercicio es configurar la Tabla de Descriptores de Interrupción (IDT) que opera en modo protegido.

Punto a)

Definimos los primeros 20 descriptores de la IDT. El tipo de descriptor que se considera para definirlos es la *interrupt gate*, la cual tiene una estructura como se puede ver en la figura 2.1.

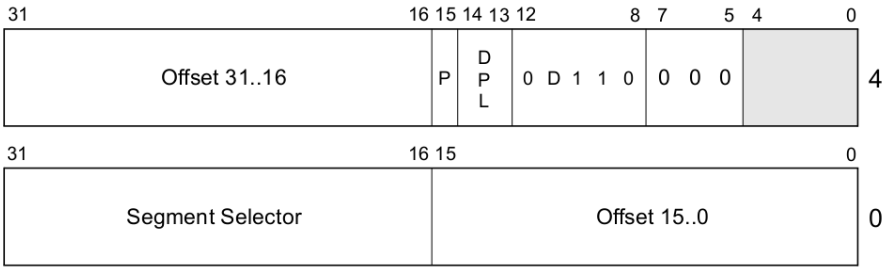

Figura 2.1: Estructura de un interrupt gate.

Además, se debe asegurar que el campo *segment selector* haga referencia al segmento de código de nivel 0 en la GDT, es decir, que su valor sea 0x40. De manera que, los descriptores se configuran de la siguiente forma:

Segment Selector	0x00000040
D	0x01
DPL	0x00
P	0x01

Cuadro 6: Descriptor de la IDT.

En todos los casos, el campo *offset* hace referencia a la dirección donde esta el código de la correspondiente interrupción.

Una vez definido estos descriptores, se carga la dirección base de la IDT en el registro IDTR mediante la instrucción LIDT.

Punto b)

Para probar que se configuró correctamente la IDT se procede a forzar una interrupción. Con ese fin, se implementa un código donde se realiza una división con cero. Se espera, entonces, ver un mensaje donde se visualice el número de interrupción que se intercepta, en este caso el número de interrupción asociada a la división por cero es 0. En la figura 2.2 se ve que esto ocurre efectivamente.

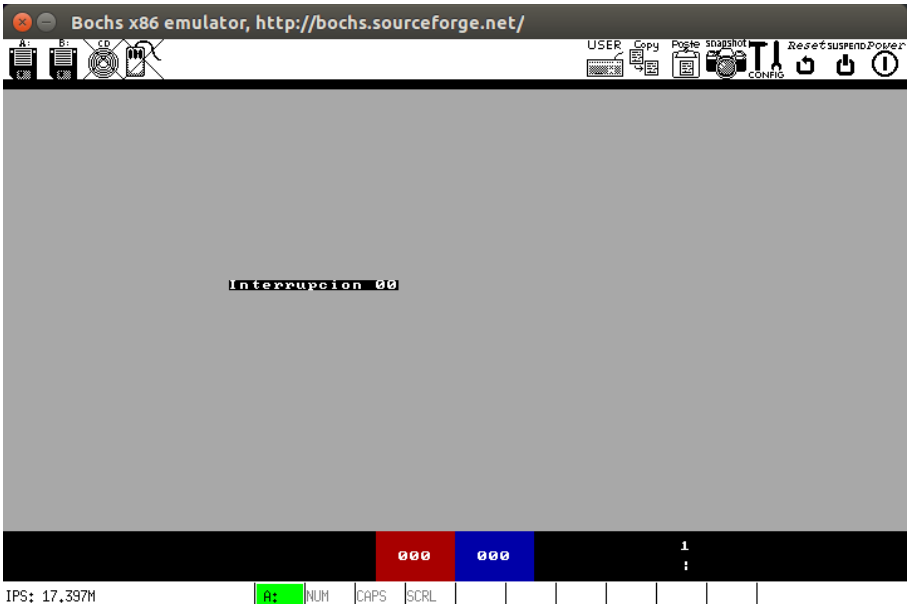

Figura 2.2: Mensaje de la interrupción 0.

3. Ejercicio 3

El objetivo de este ejercicio es activar paginación.

Punto a)

Se pinta la pantalla de como indica la figura 5 del enunciado. En la figura 3.1 se puede ver el resultado.

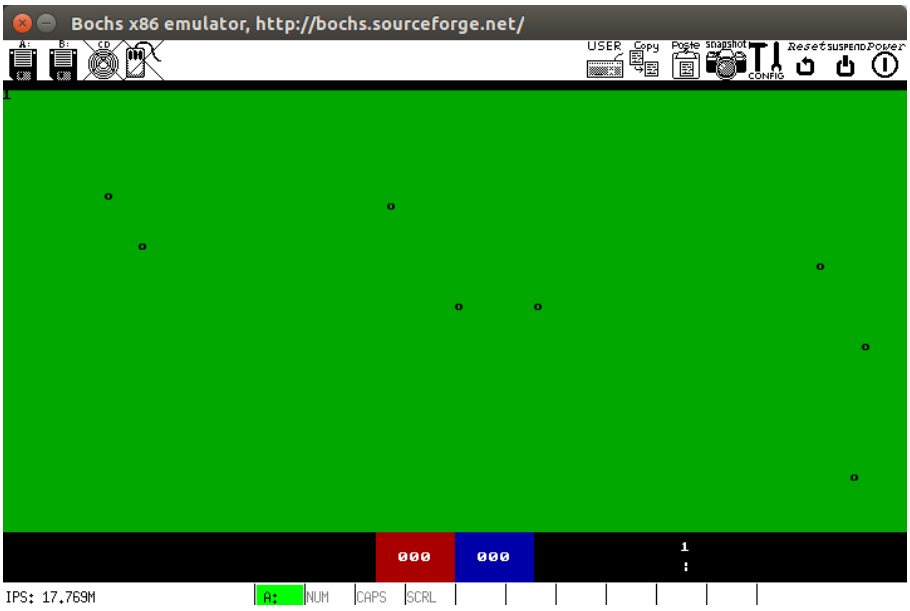

Figura 3.1: Pantalla en su estado inicial.

Punto b)

Se implementa la rutina `mmu_mapear_pagina`. Esta rutina tiene como objetivo vincular una *dirección virtual* con una *dirección física* por medio de la unidad de paginación. Los parámetros de entrada son, por lo tanto, ambas direcciones, el registro CR3 y los atributos con que se configura el descriptor de tabla de páginas. El registro CR3 debe apuntar a un directorio de tabla de páginas existente.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dirección física de la Tabla de Página																				Ignorados				PS	IGN	A	PCD	PWT	U/S	R/W	P

Figura 3.2: Descriptor de un directorio de tabla de páginas.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Dirección física del frame de la Página de 4 Kbytes																				Ignorados			G	PAT	D	A	PCD	PWT	U/S	R/W	P

Figura 3.3: Descriptor de una tabla de páginas.

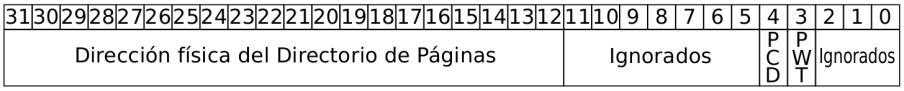

Figura 3.4: Registro CR3.

Esta rutina ejecuta los siguientes pasos. Por medio de la dirección virtual y el registro CR3 se intenta acceder al descriptor de tabla de paginas correspondiente. Si no es posible acceder entonces dicha tabla de páginas es creada inicializando sus descriptores con ceros. Una vez identificado el descriptor de tabla de páginas, se procede a registrar la dirección base de la página de 4 Kb sobre el cual apunta el cual se obtiene de los 20 bits de mayor orden de la dirección física y se configuran sus atributos con el correspondiente parámetro de entrada.

Punto c)

Se implementa la rutina `mmu_inicializar_dir_kernel` la cual esta encargada de inicializar el directorio de tabla de páginas y las tablas de páginas del *kernel*. Más precisamente, esta rutina mapea por *identity mapping* desde la dirección 0x00000000 hasta la dirección 0x003FFFFF.

Nótese que todas las direcciones virtuales mapeadas siempre se acceden al primer descriptor del directorio de tablas de páginas. También, se puede notar que con esas direcciones se puede acceder a todos los descriptores de la tabla de páginas asociada a ese primer descriptor del directorio. Teniendo en cuenta esto, la rutina mencionada al principio realiza los pasos que se mencionan a continuación.

Primero, se define el directorio de tablas de páginas a partir de la dirección 0x27000 y cada descriptor es inicializado con ceros. Luego, se define el primer descriptor del directorio apuntado a la dirección 0x28000, la cual es la base de la tabla de páginas asociada, y con los campos *R/W* y *P* configurados con el valor 1.

Luego, en la tabla de páginas ubicado en la dirección 0x28000 se configuran todos sus descriptores apuntando a la base de cada página de 0x1000 bytes empezando desde la dirección física 0x00000000 utilizando la rutina `mmu_mapear_pagina`, donde el registro CR3 apunta a la dirección 0x27000. Para cada descriptor se configuran los campos *R/W* y *P* con el valor 1.

Una vez realizado esto, se habrá mapeado las todas las direcciones desde la 0x00000000 hasta la 0x003FFFFF.

Punto d)

Para activar paginación se añade el siguiente código que realiza la actualización del bit de mayor orden del registro CR0 con el valor 1.

```
mov eax, cr0
or  eax, 0x80000000
mov cr0, eax
```

Para verificar que este paso se ha realizado correctamente se imprime, posteriormente, el nombre del grupo en la parte superior derecha de la pantalla. El resultado puede apreciarse en la figura 3.5.

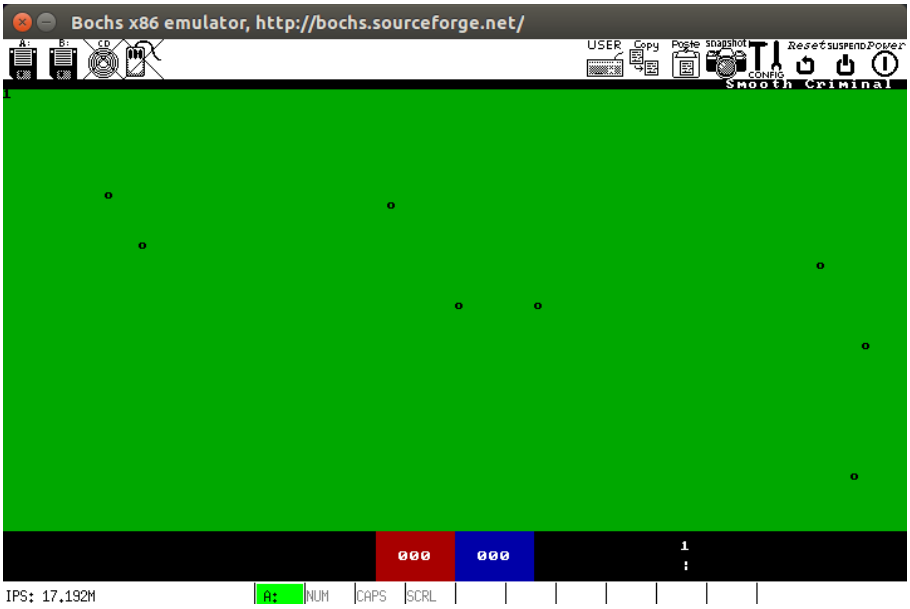

Figura 3.5: Impresión del nombre del grupo en la parte superior derecha de la pantalla.

Punto e)

Se implementa la rutina `mmu_unmappear_pagina` la cual tiene como objetivo borrar el mapeo creado para una dirección virtual en la unidad de paginación. Los parámetros de entrada requeridos son la dirección virtual y el registro CR3, el cual debe apunta a un directorio de tablas de páginas existente.

Esta rutina ejecuta los siguientes pasos. Por medio de la dirección virtual y el registro CR3 se accede al descriptor de la tabla de páginas donde se halla registrado el mapeo. Luego, es cuestión de configurar el bit de presente con 0. Si no se accede a un descriptor de tabla de páginas entonces esta rutina no realiza proceso alguno.

Punto f)

Se prueba la rutina `mmu_unmappear_pagina` desmapeando la última página del área del *kernel*, es decir, la página con dirección base 0x3FF000. Para ello, se ejecuta el siguiente código.

```
mov  eax, cr3
push eax
push 0x3FF000

call mmu_unmapear_pagina

add  esp, 8
```

4. Ejercicio 4

Punto a)

Se implementa la rutina `mmu_inicializar` la cual se encarga de inicializar las estructuras globales necesarias para administrar la memoria del área libre. Para ello, en dicha rutina, se inicializan 3 variables globales, todas ellas son direcciones en el área libre.

La primera variable, `libre`, es la dirección de la próxima página libre la cual se inicializa con la dirección base del área libre (la 0x100000). Las otras dos, `comun_a` y `comun_b`, son las direcciones de las páginas compartidas por las tareas perro de cada jugador (`comun_a` es para el jugador A y `comun_b` es para el jugador B). Para designar estas páginas se define otra rutina, sin parámetros, que devuelve el valor de la primera variable y la actualiza apuntando a la siguiente página libre. Se denominará a dicha rutina bajo el nombre de `mmu_proxima_pagina_fisica_libre`. De modo que, las dos variables restantes se inicializan con el valor que devuelve esta última rutina.

En resumen, la rutina `mmu_inicializar` inicializa 3 variables globales de la siguiente manera:

```
libre   = 0x100000;
comun_a = mmu_proxima_pagina_fisica_libre();
comun_b = mmu_proxima_pagina_fisica_libre();
```

Punto b)

Se implementa la rutina `mmu_inicializar_memoria_perro` la cual inicializa el mapa de memoria de una tarea perro. Inicializar el mapa de memoria de una tarea implica pedir páginas libre del área libre para definir el directorio y tablas de página necesarias según la figura 4.1, copiar el código de la tarea correspondiente en el área del mapa correspondiente y devolver el valor de CR3 del mapa de memoria creado, el cual contiene la dirección base del directorio de página definido.

Para pedir páginas libres del área libre se utiliza la rutina `mmu_proxima_pagina_fisica_libre` definida en el anterior punto.

Para relacionar las direcciones físicas con las virtuales se cuenta con la rutina `mmu_mapear_pagina` definida en el ejercicio 3. Se realizan los siguientes mapeos:

- Las direcciones virtuales correspondientes a los primeros 0x400000 bytes se vinculan mediante *identity mapping*.
- La dirección virtual 0x400000 se vincula con la página que comparten las tareas perro lo cual depende del jugador al que le pertenece el perro del cual se esta realizando esta incialización del mapa de memoria, es decir, se vincula con el valor que contiene la variable global `comun_a` o `comun_b` definidos en el punto anterior.
- La dirección virtual 0x401000 se vincula con la dirección física en el área del mapa donde se vaya a copiar el código de la tarea correspondiente.
- La dirección virtual 0x402000 se vincula con la dirección base física del área del mapa. Esta dirección virtual no se encuentra ilustrada en la figura 4.1 y su objetivo será revelado más adelante en este punto.
- La dirección virtual en el área del mapa donde se vaya a copiar el código de la tarea correspondiente se vincula con su respectiva dirección física en el área del mapa.

Las direcciones físicas y virtuales del área del mapa se obtienen en base una función. Sean x e y las coordenadas por columna y fila respectivamente, entonces las funciones para determinar las direcciones física y virtual son las siguientes:

```
fisica  = 0x500000 + (y * 80 + x) * 0x1000;
virtual = 0x800000 + (y * 80 + x) * 0x1000;
```

Los valores 0x500000 y 0x800000 corresponden a la dirección base física y dirección base virtual del área del mapa respectivamente (ver figura 4.1). El valor 0x1000 corresponde al tamaño en bytes de un página. El valor 80 corresponde al ancho del mapa.

Para copiar el código de la tarea perro se utiliza la dirección virtual 0x402000. Esta dirección virtual se define pues la copia de ese código se realiza en el contexto de otra tarea perro o la tarea IDLE. Además, la tarea IDLE no le corresponde a ningún jugador y no se puede hablar de un perro asociado a dicha tarea. Por ese motivo, en el área de memoria del *kernel* se agrega esta dirección virtual y se inicializa de la misma manera como se inicializa aquí. De modo que, utilizando el CR3 de la tarea actual, se vincula la dirección virtual 0x402000 con la dirección física en el área del mapa donde se debe copiar el código tarea perro correspondiente. Luego, se realiza la copia byte a byte de ese código utilizando como fuente la dirección del código de la tarea en el área del *kernel* y como destino la dirección virtual 0x402000.

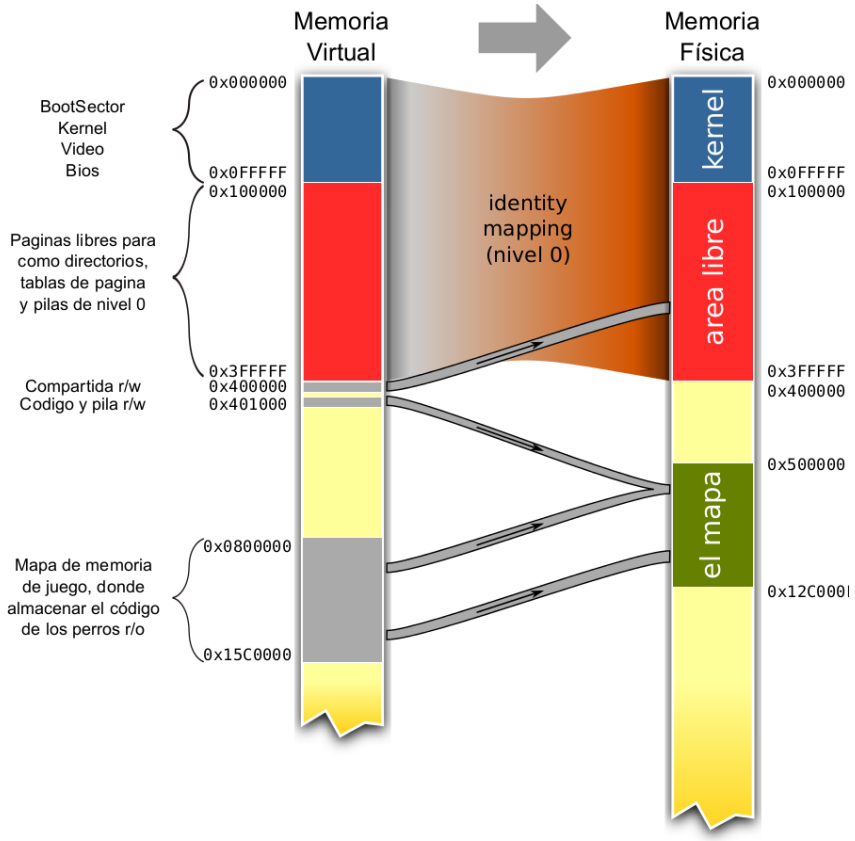

Figura 4.1: Mapa de memoria de una tarea perro.

Punto c)

Para este punto lo que se realizo fue, primero se inicializa un mapa en una tarea perro, luego se cambia el color del primer caracter y lo intercambiamos como pide el enunciado. El resultado de esto se observa en la figura 4.2.

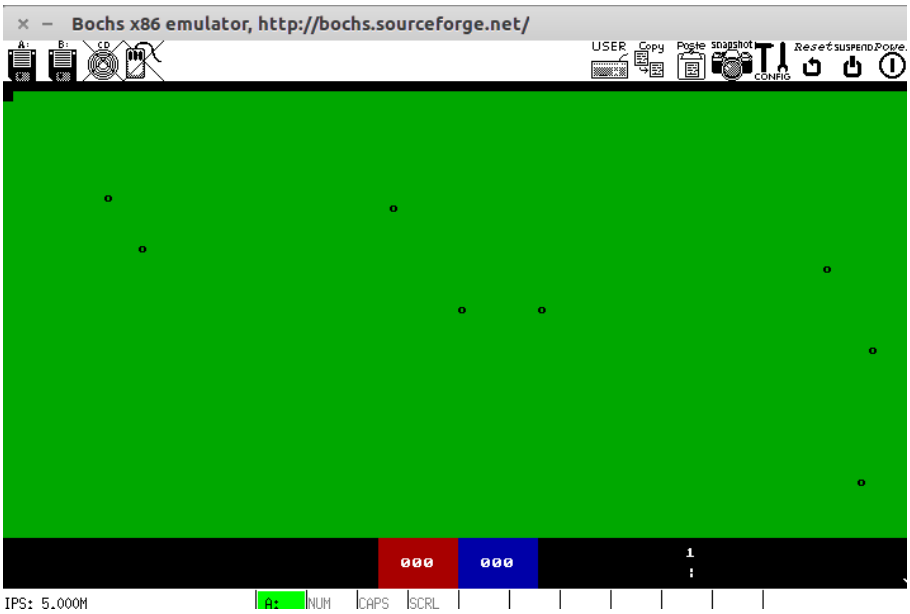

Figura 4.2: Impresión en pantalla de un caracter en el mapa para comprobar el cambio de mapa de memoria entre la del *kernel* y la de una tarea.

5. Ejercicio 5

El objetivo de este ejercicio es dar una primera configuración a las interrupciones de reloj, teclado y una que es utilizada como interrupción de software.

Punto a)

Se agregan a la IDT los descriptores correspondientes a las interrupciones de reloj, teclado y la que es utilizada como interrupción de software. La interrupción de reloj se define en el descriptor de la posición 0x20, la de teclado, en el descriptor de la posición 0x21 y la de software, en el descriptor de la posición 0x46. Todos estos descriptores son configurados similarmente a las primeras 20 interrupciones.

Para probar la funcionalidad de estas interrupciones se procede, primero, a configurar el PIC. Para ello, se cuenta con rutinas que, combinadas de la manera correcta, permiten el manejo de mas interrupciones. Las rutinas que se utilizan son `deshabilitar_pic`, `habilitar_pic` y `resetear_pic`. El orden en que estas rutinas son invocadas es la siguiente:

```
deshabilitar_pic
resetear_pic
habilitar_pic
```

Una vez realizado esto se procede a habilitar las interrupciones, es decir, configurar el bit IF del registro EFLAG con el valor 1. Para eso, se utiliza la instrucción STI.

Un detalle que debe ser mencionado en este punto es que, en cada rutina asociada a las interrupciones recientemente agregadas, lo primero que se ejecuta es la rutina `fin_intr_pic1`, la cual comunica al PIC que ya se atendió la interrupción.

Punto b)

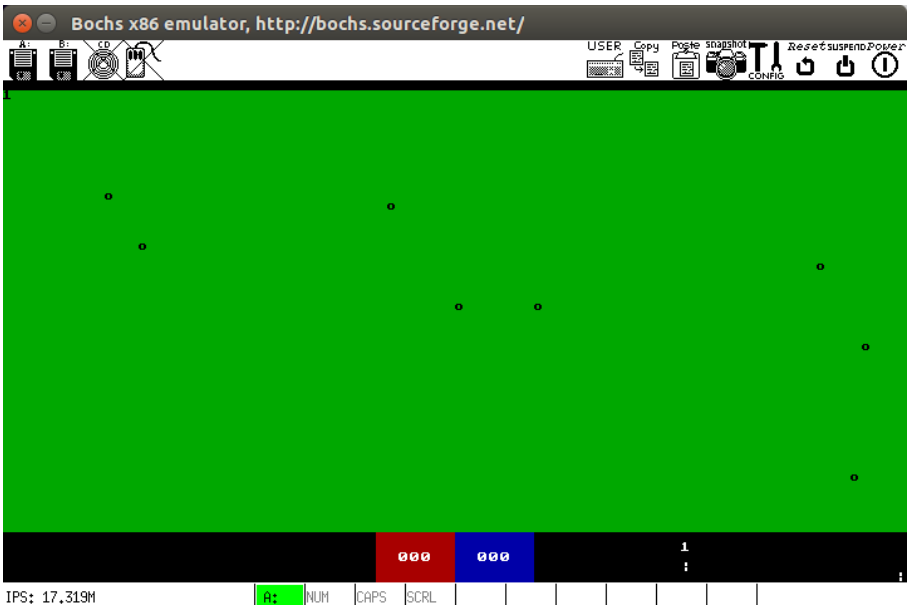

Figura 5.1: Impresión del reloj global en la parte inferior izquierda de la pantalla.

En la rutina de la interrupción del reloj, se llama a la rutina `game_atender_tick` la cual se implementa invocando a la rutina `screen_actualizar_reloj_global`, la cual ya se encuentra implementada. Esta última rutina es la imprime en la parte inferior derecha de la pantalla un cursor que, cada vez que se llame a la interrupción, irá actualizando su símbolo de modo tal que se produce el efecto en el que se visualiza a dicho cursor girando. En la figura 5.1 se puede visualizar este cursor.

Punto c)

En la rutina de la interrupción del teclado se captura el código de la tecla presionada mediante la instrucción IN de la siguiente manera:

```
in   al, 0x60
test al, 0x80
```

Una vez capturada el código de la tecla presionada, se procede a llamar a la función `game_atender_teclado` a la cual se pasa como parámetro ese código. Esta última función identifica si, efectivamente, el código corresponde a una tecla presionada y si, además, es una tecla válida del juego. Si las condiciones antes mencionadas se cumplen entonces se imprime en la parte superior derecha de la pantalla el símbolo de la tecla. En la figura 5.2 se puede visualizar el resultado de presionar la tecla W, la cual es una tecla válida del juego.

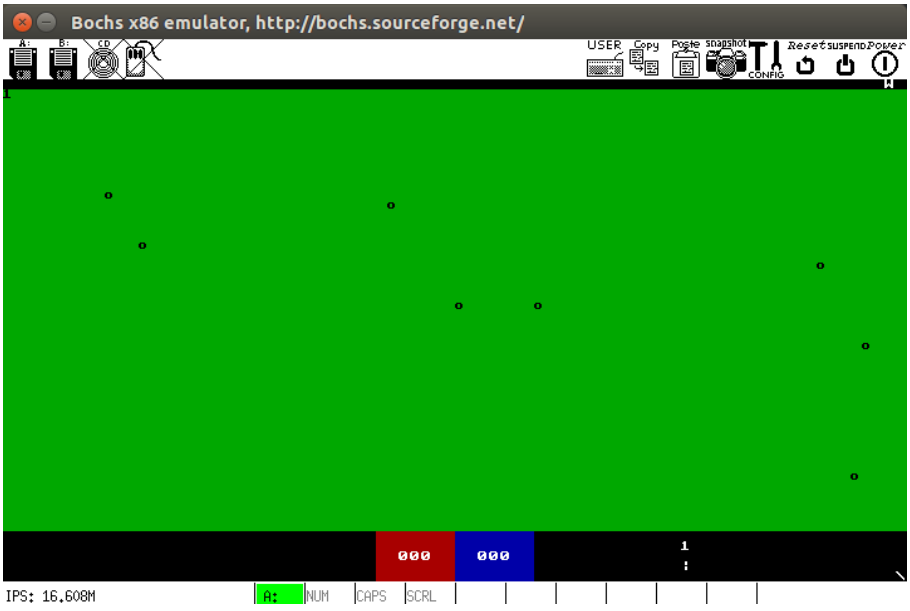

Figura 5.2: Impresión de una tecla del juego (tecla W) en la parte superior derecha de la pantalla.

Punto d)

En la rutina de la interrupción 0x46 simplemente se mueve al registro EAX el valor 0x42.

6. Ejercicio 6

El objetivo de este ejercicio es definir y cargar las tareas.

Puntos a), d) y e)

Se configuran los descriptores de la GDT de las tareas que se utilizarán. Estas tareas son la tarea inicial, la tarea IDLE y las tareas asociadas a los perros de cada jugador (16 tareas en total). Para ello, se configura cada descriptor de la GDT de la siguiente manera.

Limit	0x00067
Base	Dirección de la TSS
Type	0x09
S	0x00
DPL	0x00
P	0x01
AVL	0x00
L	0x00
DB	0x01
G	0x00

Cuadro 7: Descriptor de una tarea.

El campo *base* se configura con la dirección de la TSS de la tarea correspondiente.

Punto b)

La estructura de una TSS se puede visualizar en la figura 6.1. Se completa la TSS de la tarea IDLE de la siguiente manera:

ESP0	0x00027000
CR3	0x00027000
EIP	0x00016000
EFLAGS	0x00000202
ESP	0x00027000
EBP	0x00027000
IOMAP	0xFFFF

Cuadro 8: TSS de la tarea IDLE.

Ademas, los campos SS0, ES, SS, DS, FS, y GS se configuran como selectores de segmentos con el índice apuntando al descriptor de la GDT que describe segmento de datos de nivel 0 con RPL igual a 0. Así mismo, el campo CS se configura como selector de segmento apuntando al descriptor de la GDT que describe segmento de código de nivel 0 con RPL igual a 0. El resto de los valores se configura con el valor 0.

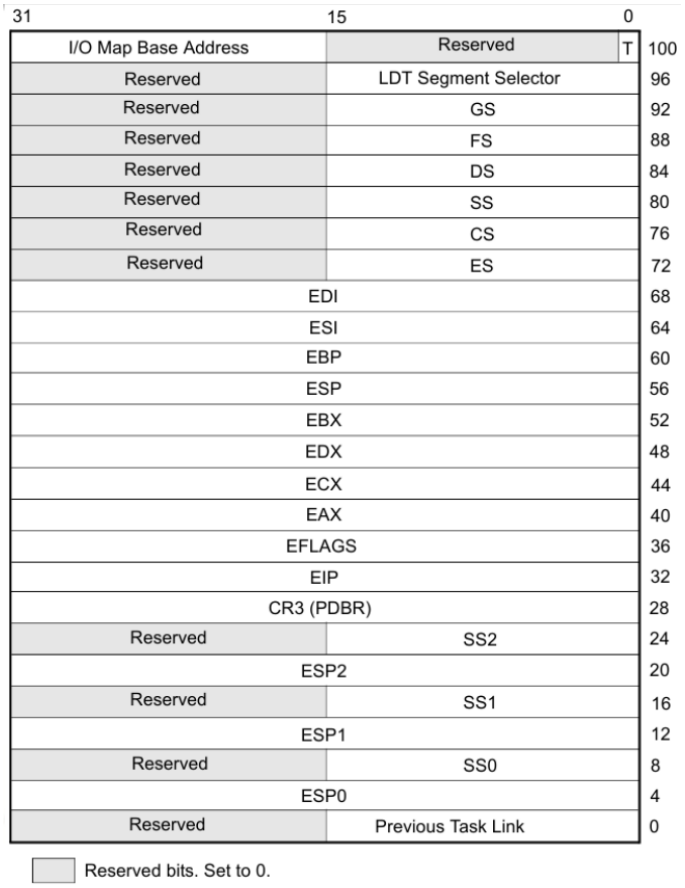

Figura 6.1: Estructura de una TSS.

Punto c)

Se completa la TSS de las tareas perro de la siguiente manera:

ESP0	`mmu_proxima_pagina_fisica_libre()`
CR3	`mmu_inicializar_memoria_perro(<perro>)`
EIP	0x00401000
EFLAGS	0x00000202
ESP	0x00402000 - 12
EBP	0x00402000 - 12
IOMAP	0xFFFF

Cuadro 9: TSS de una tarea perro.

Ademas, los campos SS0 se configura como selector de segmento con el índice apuntando al descriptor de la GDT que describe segmento de datos de nivel 0 con RPL igual a 0. Por otra parte, los campos ES, SS, DS, FS, y GS se configuran como selectores de segmentos con el índice apuntando al descriptor de la GDT que describe segmento de datos de nivel 3 con RPL igual a 3. Así mismo, el campo CS se configura como selector de segmento apuntando al descriptor de la GDT que describe segmento de código de nivel 3 con RPL igual a 3. El resto de los valores se configura con el valor 0.

Punto f)

Para saltar a la tarea IDLE primero se configura el registro TR (*Task Register*) con el selector de segmento que apunta al descriptor de la GDT que describe la tarea inicial para luego hacer un *jump far* a la ya mencionada tarea IDLE. El siguiente código ilustra lo antes comentado:

```
; Cargar tarea inicial
mov ax, 0x60
ltr ax

; Saltar a la primera tarea: Idle
jmp 0x68:0
```

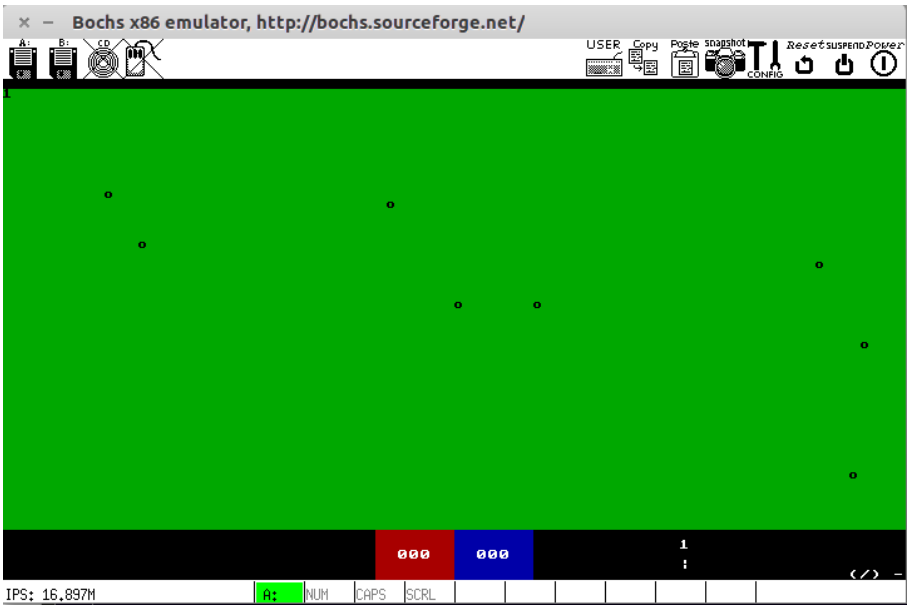

Figura 6.2: Pantalla luego de haber saltado a la tarea IDLE

Como resultado se puede ver en la pantalla un reloj, al lado del reloj global del juego (ver figura 6.2).

Punto g)

Se modifica la rutina de interrupción 0x46 para que pueda atender los servicios que puedan solicitar las tareas perro. Para ello se configuran las acciones para olfatear, cavar, moverse y recibir órdenes.

El código correspondiente a olfatear ya se encontraba implementado. Básicamente lo que haces es calcular la diferencia de la distancia que hay entre la posiciones del del perro y el escondite mas cercano. Luego, dependiendo que tan lejos según x o y se encuentre determina una dirección a seguir (arriba, abajo, derecha o izquierda).

El perro sólo realiza la acción de cavar si se encuentra parado sobre un escondite, si no, el proceso devuelve FALSE. Otra condición para que pueda cavar es si el perro no lleva consigo 10 huesos. Si es así, se devuelve FALSE. Cavar implica, por cada tick de reloj, incrementar la cantidad de huesos que puede llevar el perro en 1 y decrementar la cantidad de huesos que tiene el escondite también en 1. Si se logra cavar un hueso

entonces el proceso devuelve TRUE. En caso de poder cavar sobre un escondite y no encontrar mas huesos, entonces el proceso devuelve FALSE.

Para el perro, moverse implica actualizar su posición en la pantalla. También, implica copiar su código en una nueva posición. Para ello, primero se verifica que la posición a la cual se desea desplazar sea válida, es decir, que sea una posición del mapa. Si la posición no es válida entonces se realiza acción alguna y se termina el proceso. Copiar el código implica, primero, mapear una nueva dirección física en el mapa de memoria del perro en relación a la dirección virtual 0x401000 y la correspondiente dirección virtual del mapa. Después, se realiza la copia las direcciones virtual del mapa donde se encontraba el código anteriormente y la dirección virtual 0x401000.

Por último, recibir una orden implica conocer la última orden que da el jugador que le corresponda. Esta orden se encuentra como atributo en la estructura del jugador, por lo que para conocer dicha orden se debe acceder a ese valor, así como la posición de ese jugador, y devolverlo en el formato indicado al finalizar el proceso.

7. Ejercicio 7

El objetivo de este ejercicio es armar el *scheduler*.

Punto a)

Se implementa la rutina `sched_inicializar` la cual no recibe parámetros. Ésta rutina se encarga de inicializar las variables y estructuras globales que utiliza el *scheduler* para administrar las tareas a ejecutar. Dichas variables y estructuras son las siguientes:

- `tareas`: Un array con información correspondiente a todas las tareas. Cada elemento del array contiene la posición en la GDT donde se halla el descriptor de dicha tarea y un puntero, en caso de ser posible, a la información del perro asociado.
- `tarea_actual`: Un entero que indica cual es la tarea que se esta ejecutando actualmente.
- `tarea_a`: Un entero que indica cual es la tarea del jugador A que se esta ejecutando actualmente. Si no se esta ejecutando una tarea perro del jugador A entonces el valor de esta variable es -1.
- `tarea_b`: Un entero que indica cual es la tarea del jugador B que se esta ejecutando actualmente. Si no se esta ejecutando una tarea perro del jugador B entonces el valor de esta variable es -1.

Todos estas estructuras se inicializan como se describe a continuación:

- `tareas`: el elemento de la posición 0 del array corresponde a la tarea IDLE, el cual no tiene perro asociado y, por lo tanto, el valor del puntero es NULL. Los elementos de las posiciones 1 a 8 del array corresponden a las tareas perro del jugador A. Y los elementos de las posiciones 9 a 16, a las tareas perro del jugador B.
- `tarea_actual`: se inicializa con el valor 0 pues se desea que la tarea IDLE sea la primera en ser ejecutada.
- `tarea_a`: se inicializa con -1 debido a que la tarea IDLE es la primera en ser ejecutada.
- `tarea_b`: se inicializa con -1 debido a que la tarea IDLE es la primera en ser ejecutada.

Punto b)

Se implementa la rutina `sched_proxima_a_ejecutar` la cual no recibe parámetros. Ésta rutina se encarga de determinar la tarea que debe ejecutarse y para ello devuelve la posición en la GDT del descriptor de dicha tarea. Para ello, en pocas palabras, se itera sobre los elementos del array de tareas y se busca la primera tarea perro que este viva dependiendo a que jugador le corresponda la tarea que se esta ejecutando actualmente.

La política elegida para la búsqueda de la próxima tarea entre las tareas de ambos jugadores es la siguiente. Primero se determina cual es la tarea actual y, luego, dependiendo a que jugador le corresponda dicha tarea se dispone a buscar entre las tareas vivas del otro jugador. Si no se encuentra una tarea viva entre las tareas del otro jugador, ya sea porque no tiene o porque ya se han recorrido todas las tareas vivas, entonces se vuelve a buscar entre las tareas del jugador al que le pertenece la tarea que se esta ejecutando actualmente. El caso particular se produce cuando la tarea actual es la IDLE pero este caso se trata como si la tarea actual perteneciese al jugador B. Por ejemplo, si la tarea que se estaba ejecutando actualmente es una tarea viva del jugador B entonces se busca entre la tareas vivas del jugador A, pero si no se encuentra tarea viva entre

las tareas del jugador A entonces se vuelve a buscar entre las tareas vivas del jugador B. En este contexto, puede ocurrir que no haya tarea viva a ejecutar por parte de ambos jugadores. En ese caso, la posición que se devuelve es la asociada a la tarea IDLE.

Para entender mejor el funcionamiento de esta rutina se presenta el pseudocódigo del algoritmo que implementa la rutina:

```
// Se determina la próxima tarea a ejecutar
Function sched_proxima_a_ejecutar()
    if tarea_actual > 0 y tarea_actual < 9 then
        // Si la tarea actual es una tarea del jugador A
        i ← sched_buscar_tarea_libre_b()
        if i ≠ −1 then
            // Es el turno de una tarea del jugador B
            return i
        end
        i ← sched_buscar_tarea_libre_a()
        if i ≠ −1 then
            // Es el turno de una tarea del jugador A
            return i
        end
    else
        // Si la tarea actual es una tarea perro del jugador B o es la tarea IDLE
        i ← sched_buscar_tarea_libre_a()
        if i ≠ −1 then
            // Es el turno de una tarea del jugador A
            return i
        end
        i ← sched_buscar_tarea_libre_b()
        if i ≠ −1 then
            // Es el turno de una tarea del jugador B
            return i
        end
    end
    // Es el turno de la tarea IDLE
    return 0
```

Algoritmo 1: Algoritmo implementado por la rutina `sched_proxima_a_ejecutar`.

Los algoritmos que encierran las rutinas `sched_buscar_tarea_libre_a` y `sched_buscar_tarea_libre_b` se describen a continuación:

```
// Próxima tarea libre del jugador A
Function sched_buscar_tarea_libre_a()
    tarea_a ← tarea_a + 1
    i ← tarea_a
    while i < 8 do
        if tareas[1 + i] es una tarea viva then
            tarea_a ← i
            return 1 + i
        end
        i ← i + 1
    end
    tarea_a ← −1
    return tarea_a
// Próxima tarea libre del jugador A
Function sched_buscar_tarea_libre_b()
    tarea_b ← tarea_b + 1
    i ← tarea_b
    while i < 8 do
        if tareas[9 + i] es una tarea viva then
            tarea_b ← i
            return 8 + i
        end
        i ← i + 1
    end
    tarea_b ← −1
    return tarea_b
```

Algoritmo 2: Algoritmos de rutinas invocadas por `sched_proxima_a_ejecutar`.

Punto c)

En el presente trabajo las rutinas `game_atender_tick` y `sched_atender_tick` son procesos independientes entre si, es decir, un proceso no invoca al otro.

La rutina `game_atender_tick` utiliza el valor de una variable global que indica cual es el perro que se esta ejecutando actualmente, llámese `perro_actual`, para poder realizar acciones relacionadas a la parte visual del juego como, por ejemplo, imprimir los relojes del perro por pantalla, mostrar el cartel de fin del juego, mostrar la pantalla del modo debug, entre otros.

En cambio, la rutina `sched_atender_tick` se encarga de elegir la próxima tarea a ejecutar dependiendo del contexto del juego y devuelve el selector de segmento con el cual se debe realizar el salto a dicha tarea. Para determinar la próxima tarea a ejecutar utiliza la rutina `sched_proxima_a_ejecutar` descrita en el punto anterior. Para determinar si elegir o no la próxima tarea a ejecutar utiliza el valor de otra variable global del juego que indica si un perro dejo de vivir por causas naturales, llámese `exploto_algo`, lo cual se produce cuando el código de dicho perro produce una excepción. Así, si se confirma que esa variable es cierta entonces no se busca una próxima tarea y, por lo tanto, el selector de segmento que se devuelve es el mismo que se devolvió en la anterior llamado a esta rutina (`sched_atender_tick`), el cual es, justamente, el selector asociado a la tarea perro que provocó la excepción. Esta acción la seguirá realizando hasta que el valor de `exploto_algo` cambie a su valor opuesto y pueda determinar una nueva tarea, si es que la hay. Además, la rutina en cuestión también determina el valor de la variable `perro_actual`, presentada en el anterior párrafo, ya que el array de tareas contiene esa información. Este funcionamiento se entenderá

mejor cuando se describa la interrupción de atención del reloj.

Por último, se menciona que el selector de segmento que se devuelve tiene como valor de RPL a 0 y como valor de IT a 0.

Punto d)

La rutina de interrupción 0x46 se modifica agregando, luego de atender la petición del la tarea perro, un salto a la tarea IDLE.

Por otro lado, la rutina de interrupción del teclado se modifica agregando un filtro que depende del valor de `exploto_algo`, presentada en el punto anterior. Si `exploto_algo` es cierto entonces no se debe atender ninguna petición del teclado salvo la tecla Y cuando es presionada. Esto esta fuertemente ligado al modo debug lo cual se explicará más adelante. Si el valor de `exploto_algo` es falso entonces se permite la atención de las teclas permitidas por el juego.

Punto e)

La rutina de atención del reloj se actualiza de la siguiente manera:

```
sched_tarea_offset:     dd 0x00
sched_tarea_selector:   dw 0x00

_isr32:
    pushad
    pushfd

    call fin_intr_pic1

    call game_atender_tick
    call sched_atender_tick

    str cx
    cmp ax, cx

    je .fin
        mov [sched_tarea_selector] , ax
        jmp far [sched_tarea_offset]
    .fin:

    popfd
    popad

    iret
```

A partir de esto se puede decir lo siguiente. El *quantum* del *scheduler* es 1 ya que se invoca a la rutina `sched_atender_tick` por cada interrupción de reloj. También, si `exploto_algo` es cierta, el selector de segmento que se obtiene de `sched_atender_tick` es el mismo que se obtuvo en la anterior interrupción de reloj y, por lo tanto, no se produce el salto a la tarea, de modo que cuando se ejecute la sentencia IRET se abandona la rutina de atención al reloj y se vuelve al código que se estaba ejecutando antes, la cual no es nada más ni nada menos que la tarea IDLE (este comportamiento se entenderá mejor cuando se explique el funcionamiento del modo debug).

Punto f)

Las rutinas de las excepciones 0 a 19 se modifican de la siguiente manera. Primero se obtiene la información del perro que se esta corriendo actualmente, lo cual se obtiene del array de tareas con ayuda de `tarea_actual`. La rutina `sched_tarea_actual` devuelve la información antes descrita. Luego, se actualiza la variable de ese perro que indica que esta vivo para indicar que ya no lo está. Con esto último es más que suficiente para desalojar dicha tarea del *scheduler* y la rutina `game_perro_termino` lleva a cabo este proceso. Por último se realiza el salto a la tarea IDLE.

En el siguiente código se puede apreciar mejor lo descrito en este punto.

```
; se desaloja la tarea actual que es la que provoco esta interrupcion
call sched_tarea_actual

push eax
call game_perro_termino
add  esp, 4

; se salta al a tarea IDLE
jmp 0x68:0
```

Punto g)

El modo debug se activa y desactiva a través de la interrupción de teclado. Cuando dicha interrupción recibe la tecla Y se actualiza una variable global que indica que el modo debug esta activo o desactivado, llámese `debug_time`, dependiendo del valor anterior, es decir, si se encontraba activo entonces se actualiza a desactivo y viceversa. Inicialmente, `debug_time` se encuentra con el valor que indica que el modo debug se encuentra desactivado.

Se agrega a las rutinas de excepción la invocación a un proceso que activa otra variable global indicando que debe la ventana del modo debug. Esta variable es, justamente, `exploto_algo`, y sólo es actualizada si el modo debug esta activo. También, al momento de actualizar `exploto_algo`, se guarda el estado actual de la pantalla. Cuando el modo debug es desactivado, presinando de vuelta la tecla Y, se utiliza ese estado guardado para recuperar la pantalla anterior.

Cuando se produce una excepción en modo debug, se debe recuperar la información del estado de ciertos registros antes de producirse dicha excepción en su rutina de atención. Debido a que siempre se produce un cambio del privilegio cuando se producen estas excepciones, pues las excepciones se producen en nivel usuario y son tratadas en nivel supervisor, muchos de esos registros se encuentran en la pila, tales como los selectores CS, SS, ESP, EIP y EFLAGS (ver figura 7.1). Pero para recuperar estos valores, hay que tener en cuenta si al momento de producirse la excepción se apila el *error code*. Es por ello que, dependiendo de este detalle, se recuperan los valores que se muestran en la ventana del modo debug.

Luego, cuando se produzca la próxima interrupción de reloj, en la rutina `game_atender_tick`, se muestra la ventana del modo debug dado que `debug_time` indica que le modo debug esta activo y `exploto_algo` indica que se produjo una excepción debido a un error en el código de una tarea perro.

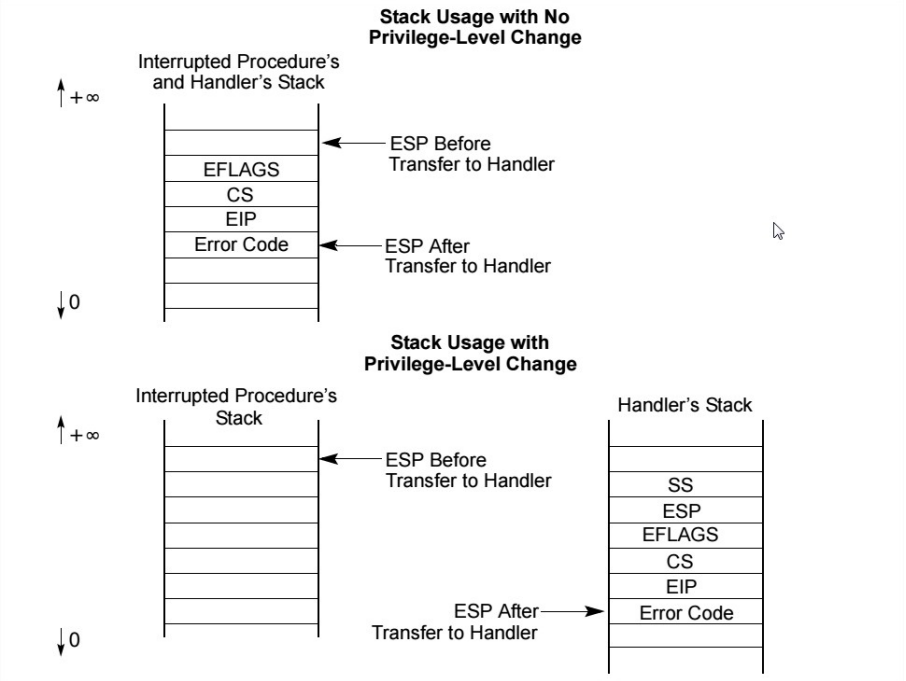

Figura 7.1: Pantalla luego de haber saltado a la tarea IDLE