DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

21 de julio de 2017

Organización del Computador II
Primer Cuatrimestre de 2017

Grupo "InSisto gEnio zen acaba"

Integrante	LU	Correo electrónico
Bonggio, Enzo	074/15	ebonggio@dc.uba.ar
Szperling, Sebastián Ariel	763/15	sszperling@dc.uba.ar
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Ejercicio 1: GDT, Segmentos y Modo Protegido

1.1. Descriptores de segmentos

Por las restricciones pedidas en el TP, nuestros descriptores comienzan en índice 8 de la GDT. El esquema utilizado es de segmentación *flat*, con la excepción del segmento de video que solo es utilizado al dibujar el mapa por primera vez. PAra cumplir los requisitos, se definieron los segmentos como descriptores en la GDT de la siguiente manera:

Índice	Base	Límite	DPL	Tipo	G
8	0x000000	0x26EFF	0x0	0xA	1
9				0x2	
10			0x3	0xA	
11				0x2	
15	0xB8000	0x13F3	0x0	0x2	0

Los niveles de privilegio corresponden a kernel (`0x0`) y a usuario/tarea (`0x3`), mientras que los tipos corresponden a código con lectura (`0xA`) y a datos con lectura/escritura (`0x2`).

Todos los segmentos, además, están marcados como presentes (`p=1`) y de 32 bits (`db=1`). Los primeros 4 segmentos abarcan en total 623MB, mientras que el segmento de video solo ocupa lo necesario para escribir el buffer en su totalidad ($80 \times 50 \times 2$ bytes).

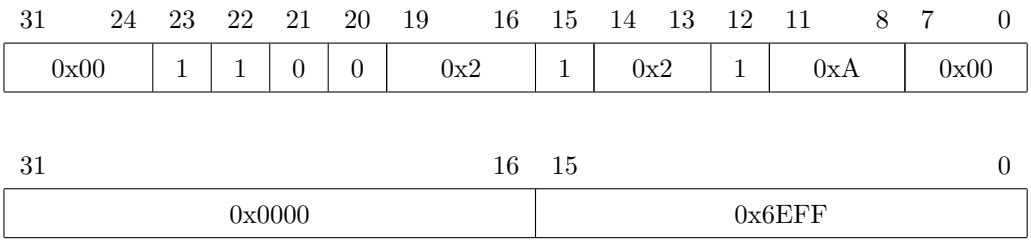

Figura 1: Ejemplo de descriptor de GDT: segmento de código de kernel

Dado que nuestros segmentos ocupan 623MB, antes de cargar la GDT debimos activar la linea A20.

1.2. Salto a modo protegido

Para pasar de modo real a modo protegido y configurar la pila del kernel, debimos:

1. Setear el bit CR0.PE (bit 0):

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

2. Inmediatamente realizar un jump far con selector 0x40 (índice 8 de la GDT, código de kernel):

```
jmp 0x40:modoprotegido
```

3. Configurar los registros de selectores con 0x48 (índice 9 de la GDT, datos de kernel), salvo por `fs` que fue seteado a 0x78 (índice 15 de la GDT, segmento de video):

```
mov ax, 1001000b ; index = 9 | gdt = 0 | rpl = 0
mov ds, ax
mov es, ax
mov gs, ax
mov ss, ax
mov ax, 1111000b ; index = 15 | gdt = 0 | rpl = 0
mov fs, ax
```

4. Por úlitmo, setear la pila a la dirección correspondiente:

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

2. Ejercicio 2: IDT y rutinas de atención a excepciones

2.1. Inicialización

Para poder tener acceso a la *IDT* se requiere de un descriptor que indique el lugar donde se encuentra el arreglo con las entradas de atención a interrupciones.

```
idt_descriptor IDT_DESC = {
        sizeof(idt) - 1,
        (unsigned int) &idt
};
```

Las entradas de la *IDT* fueron generadas mediante la utilización del macro provisto por la cátedra, el cual especificaba que la atención de la misma iba a darse en el archivos *isr.asm* en el método llamado *_isrX* donde *X* representa el numero de interrupción a responder. Se completaron todas las entradas correspondientes a las excepciones definidas por Intel, además de las interrupciones de reloj (32) y teclado (33) y la puerta de servicio de sistema (102 o `0x66`).

Todas utilizan el selector de segmento de código del kernel (`0x40`) y cuentan con atributo `0x8E00` (interrupt gates presentes de nivel 0), salvo por la interrupción `0x66` que debe ser accesible desde nivel de privilegio usuario (3), por lo que su atributo se modifica a `0xEE00`.

Una vez llenados los campos cargamos la IDT a través de la instrucción `lidt [IDT_DESC]`.

2.2. Excepciones

En el caso de una excepción, nuestro sistema debe desalojar la tarea actual y saltar nuevamente a la tarea Idle. Para atender las excepciones usamos el macro provisto por la cátedra:

```
%macro ISR 1
global _isr%1

_isr%1:
    push %1
    jmp matar_tarea

%endmacro

ISR 0; inicializamos los macros que eran necesarios para atender las excepciones
...
```

En cuando al código de manejo de la excepción agregamos la lógica de matar a la tarea y saltar a idle. A lo que nos referimos cuando hablamos de matar a la tarea:

1. Imprimir en pantalla la muerte del zombi.

   ```
   call game_matar_zombi_actual
   ```

2. Chequear si las condiciones para que el juego finalice se cumplen. Es decir ver si aun quedan zombis por tirar.

   ```
   if(jugadores[JUG_A].remaining == 0 && jugadores[JUG_B].remaining == 0 &&
                   jugadores[JUG_A].current == 0 && jugadores[JUG_B].current == 0) {
           game_finalizar();
   }
   ```

3. En caso de estar en modo debug guardo y muestros la información anteriormente guardada.

   ```
   mov al, [debug_flag]
   test al, debug_on
   jz .tarea_muerta

   pushad

   sub esp, 4 ; espacio para eip

   mov eax, cr0
   push eax
   mov eax, cr2
   push eax
   mov eax, cr3
   push eax
   mov eax, cr4
   push eax
   ```

```
push cs ; push segmentos
push ds
push es
push fs
push gs
push ss

push esp
```

4. Indicarle al scheduler que no conmute mas tareas.

```
call sched_toggle_debug ; dejar de conmutar tareas
```

5. Sacar la tarea de la lista del scheduler y por ultimo saltar a idle.

```
call sched_matar_tarea_actual
```

De esta manera al próximo ciclo de clock el scheduler se encargara nuevamente de buscar la tarea correspondiente.

3. Ejercicio 3: Paginación y mapa de páginas del Kernel

3.1. Mapeo de páginas del kernel

El kernel ocupa 1MB de memoria, con 3MB adicionales de area libre. Por ende, este area exactamente en el espacio de una tabla de páginas completa.

Dado que todas las tareas deben tener este area mapeada igual que el kernel (con *identity mapping*), creamos una función genérica para mappear el kernel que toma un page directory y un page table como parámetros:

```
void mmu_mapear_dir_kernel(unsigned int pd, unsigned int pt) {
        pd_entry* dir_paginas = (pd_entry*) pd;
        dir_paginas[0].p = 1; // bit presente
        dir_paginas[0].rw = 1; // incluye codigo y datos, read/write
        dir_paginas[0].us = 0; // area de kernel - privilegio supervisor
        dir_paginas[0].pwt = 0;
        dir_paginas[0].pcd = 0;
        dir_paginas[0].a = 0;
        dir_paginas[0].ign = 0;
        dir_paginas[0].ps = 0;
        dir_paginas[0].g = 0;
        dir_paginas[0].avl = 0;
        dir_paginas[0].page_addr = pt >> 12; // apunta a la base de la tabla de paginas

        pt_entry* tabla = (pt_entry*)pt;
        unsigned int i;
        for(i = 0; i < 1024; i++) {
                // mismos atributos que en la PDE
                tabla[i].p = 1;
                tabla[i].rw = 1;
                tabla[i].us = 0;
                tabla[i].pwt = 0;
                tabla[i].pcd = 0;
                tabla[i].a = 0;
                tabla[i].d = 0;
                tabla[i].pat = 0;
                tabla[i].g = 0;
                tabla[i].avl = 0;
                tabla[i].page_addr = i; // igual que el indice para identity mapping
        }
}
```

Esto asegura que todo el area del kernel está presente y mapeada por *identity mapping* con permisos de supervisor. De este modo, para configurar los mapeos del kernel, alcanza con llamar a:

```
void mmu_inicializar_dir_kernel() {
        mmu_mapear_dir_kernel(0x27000, 0x28000);
}
```

siendo estas las direcciones del directorio y la tabla de páginas del kernel definidas por la cátedra.

3.2. Habilitado de paginación

Para habilitar el sistema de paginación, debimos:

1. Inicializar el directorio de páginas del kernel como se explicó en el punto anterior:

   ```
   call mmu_inicializar_dir_kernel
   ```

2. Cargar el directorio de páginas al registro CR3:

   ```
   mov eax, 0x27000
   mov cr3, eax
   ```

3. Setear el bit CR0.PG (bit 31):

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

4. Ejercicio 4: Unidad de Manejo de Memoria

4.1. Inicialización

En la inicialización de la MMU, simplemente inicializamos el valor de la variable que nos dará la próxima página libre. Según el mapa de memoria de la cátedra, esta variable es inicializada al comienzo del area libre, ubicada en `0x100000`. Cada vez que se requiera una página libre, se retornará este valor y será avanzado en `0x1000` (para apuntar a la siguente página).

Este area está mapeada con *identity mapping*, por lo que no es necesario agregar las nuevas páginas al mappeo.

4.2. Mapeo de tareas "zombi"

En la inicialización del mapeo de una tarea, debemos:

- pedir dos páginas libres para utilizarlas como directorio y tabla de páginas y usarlas para todos los mapeos siguientes;
- mapear el area del kernel, correspondiende a los primeros 4MB de memoria;
- mapear el area del mapa en la que la tarea será copiada, que corresponde a la página de la tarea misma más sus casilleros aledaños;
- copiar el código de la tarea a su posición en el mapa, que requiere un mapeo temporal por *identity mapping* en el directorio de páginas activo (es decir, el CR3 actual)

Para el mapeo del kernel podemos utilizar la función definida para el ejercicio 3, ya que el mapeo es idéntico (incluyendo los permisos de supervisor).

Para realizar los mapeos del mapa, tanto los temporales como los que corresponden a la nueva tarea, se usan funciones que definimos a continuación para mapear páginas puntuales.

En cuanto al area alrededor de la tarea/zombi:

- al calcular tanto la posición inicial como las posiciones relativas, se debe tener en cuenta el jugador que lanza la tarea:

```
char direccion = jugador == JUG_A ? 1 : -1;

unsigned int centro = yPos * MAP_MEM_WIDTH;
if(jugador == JUG_B) {
        centro += MAP_MEM_WIDTH - PAGE_SIZE;
}
```

- en el mapeo inicial solo se mapean 5 de los 8 casilleros adyacentes (ya que el zombi se encuentra al borde del mapa, no tiene sus casilleros "anteriores");
- para las posiciones superiores e inferiores (o izquierda y derecha del zombi) se debe tener en cuenta el caso en que la dirección excede los límites del mapa, y debe mapearse el extremo opuesto en caso contrario; por ejemplo:

```
mmu_mapear_pagina(TASK_VIRT+(2*PAGE_SIZE), pd,
        MAP_START + mem_mod(centro + direccion * (PAGE_SIZE + MAP_MEM_WIDTH), MAP_MEM_SIZE));
```

  donde la función `mem_mod` simplemente calcula el módulo entre dos números (a diferencia del operador `%` que puede dar resultados negativos).

4.3. Mapeo general de páginas

Para el mappeo de páginas definimos 2 macros que nos permiten calcular los offsets en el directorio y la tabla correspondientes:

```
#define PDE_OFFSET(virtual) virtual >> 22
#define PTE_OFFSET(virtual) (virtual << 10) >> 22
```

Por otro lado, creamos 2 funciones para crear o destruir mappeos para usuario:

- Para la creación de un mapeo, primero se revisa la PDE correspondiente: si la tabla no está marcada como presente, se pide una nueva página para esta ser utilizada como tabla de páginas y se la almacena como presente en la PDE mencionada. Luego, en esta tabla se accede a la entrada correspondiente y se guarda la base de la dirección física con atributos de lecto-escritura y nivel de privilegio usuario y el bit de presente.

- Para desmappear, el procedimiento es muy similar, pero tras conseguir la dirección de la PTE, basta con limpiar el bit de presente en la misma; si la tabla de páginas no existe (no está presente), la función no hace nada.

Luego de ambas funciones se llama a la función `tlbflush()`, que se encarga de limpiar el caché de traducciones de direcciones (*Translation Lookaside Buffer*) para que el cambio se vea reflejado en caso de tratarse del CR3 actual.

Cabe aclarar que las tareas necesitan mapear 1 página con privilegios de supervisor (la pila de nivel 0 correspondiente a la misma), por lo cual esto se corrige luego del mapeo. Esto no genera conflictos con el caché de traducciones, ya que esta página se mapea una única vez durante la creación de la tarea (durante la cual su CR3 no es el activo).

5. Ejercicio 5: Interrupciones de reloj y teclado

EStas dos interrupciones tienen 2 detalles en común: por un lado, deben ser ignoradas si el juego finalizó, lo cual se señaliza con la dirección de memoria ENDGAME (fin de juego); por el otro, en ambos casos debemos reiniciar el PIC al finalizar la atención a la interrupción para poder atender fururas interrupciones.

5.1. Rutina de atención a reloj

Dentro de la rutina del reloj se encuentra el código encargado de la conmutación de tareas:

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

_isr32:
    pushad
    test byte [ENDGAME], 1
    jnz .end

    call proximo_reloj

    call sched_tarea_actual
    cmp eax, 16
    jge .next_task
    push eax
    call game_print_clock
    pop eax

    .next_task

    call sched_proximo_indice
    cmp ax, 0
    je .nojump
        mov [sched_tarea_selector], ax
        call fin_intr_pic1
        jmp far [sched_tarea_offset]
        jmp .end

    .nojump:
    call fin_intr_pic1

    .end:
    popad
    iret
```

Para lograr saber a que tarea saltar debe llamarse al scheduler, explicado en mayor detalle en el ejercicio 7, el cual nos devuelve el selector de la próxima tarea a ejecutar. En caso de que no haya que realizar un cambio de tarea (no hay tareas o se trata de la tarea actual) volvemos con *IRET*.

En este punto también actualizamos el reloj de la tarea que se ejecuta actualmente en la pantalla, además del reloj global del sistema. Si el juego terminó, el sistema deja de conmutar tareas.

5.2. Rutina de atención a teclado

Las interrupciones del teclado nos dan la posibilidad de poder elegir y lanzar tareas, pero también nos permiten activar y desactivar el modo debug del cual daremos mas detalles en el ejercicio 7.

Tenemos en cuenta dos eventos a la hora de activar o desactivar la posibilidad de lanzar tareas o elegirlas, estas son:

- el final de juego la cual anula por completo la posibilidad de lanzar o cambiar el zombi.
- y ademas cuando el cartel de debug se esta mostrando no se permite hacer ninguna acción, salvo desactivar el cartel para poder seguir jugando.

No se tomo en cuenta los códigos de tecla para cuando uno suelta una tecla, ya que solo consideramos como casos a resolver por la interrupción los códigos de las teclas *A, S, W, X, L_Shift* para el `jugador A` y *J, K, I, M, R_Shift* para el `jugador B`, ademas contamos con la tecla *Y* para activar el modo debug caso contrario la interrupcion no realiza ninguna acción.

Si el juego terminó, el input del usuario debe ser ignorado, y el sistema debe ser reiniciado.

6. Ejercicio 6: TSSs y salto a tarea Idle

6.1. Entradas en la GDT

Para saltar a las distintas tareas, definimos un total de 18 entradas en la GDT correspondientes a cada TSS. Las mismas tienen todas los mismos atributos: son de sistema, tipo 0x9, con límite 0x68. La dirección base de cada TSS está definida por 2 posiciones constantes (para la inicial y la Idle) y 2 dos arreglos (para las tareas zombi).

Para que todas las entradas entren cómodamente en la GDT, aumentamos su tamaño a 32. Sin embargo, podría reducirse a 30 si 1) usamos un espacio en blanco que dejamos para linear los números y 2) utilizamos la TSS de una tarea zombi como TSS inicial.

6.2. Inicialización de las TSSs

La TSS de la tarea inicial no hace falta inicializarla ya que solo utiliza como dummy para poder saltar a la "siguiente"(primera) tarea, la Idle. Por ende, sus valores nunca son leídos, solo escritos.

La TSS Idle se inicializa con los siguientes valores:

- `CS = 0x40` (segmento de código del kernel)
- `DS, ES, FS, GS, SS, SS0 = 0x48` (segmento de datos del kernel)
- `ESP, EBP, ESP0 = 0x27000` (base de pila del kernel)
- `CR3 = 0x27000` (directorio de páginas del kernel)
- `EIP = 0x10000` (base de la tarea Idle)

En cuanto a las tareas zombi, sus TSSs se inicializan de la siguiente manera:

- `CS = 0x53` (segmento de código de usuario, con DPL 3)
- `DS, ES, FS, GS, SS = 0x5B` (segmento de datos de usuario, con DPL 3)
- `ESP, EBP = 0x08001000` (límite de página virtual de la tarea)
- `SS0 = 0x48` (segmento de datos del kernel)
- `ESP0 = 0x0800b000` (límite de dirección virtual mapeada a pagina nueva)
- `CR3 = mmu_inicializar_dir_zombi` (directorio creado durante la inicialización, ver sección 4.2)
- `EIP = 0x08000000` (base de página virtual de la tarea)

La base de la página designada para el stack de nivel 0 (ESP0) se ubica en `0x0800a000`.

Para todas las tareas, los registros generales se inicializan en 0, el mapa de I/0 en `0xFFFF` y los `EFLAGS` en `0x202` (interrupciones activadas).

6.3. Salto a tarea Idle

Para saltar a la primera tarea debimos:

1. Inicializar las TSSs y agregarlas a la GDT:

```
call tss_inicializar
call tss_inicializar_idle
```

2. Cargar la TSS inicial como "tarea actual":

```
mov ax, 0x60
ltr ax
```

3. Hacer un jump far (task switch) a la tarea Idle:

```
jmp 0x68:0
```

7. Ejercicio 7: scheduling, servicio mover y modo debug

7.1. Inicialización del scheduler

Para hacer funcionar el scheduler debimos asegurarnos de tener las interrupciones activadas con el PIC configurado para poder recibir las interrupciones del reloj. Esto se inicializa antes de saltar a la tarea Idle por primera vez.

Adicionalmente, el scheduler se inicializa con tareas inválida, ya que el mismo mantiene el número de la última tarea ejecutada para cada jugador. Los valores inválidos omiten ciertos procesos como el dibujado del reloj de la tarea actual (la tarea Idle maneja su propio reloj).

7.2. Conmutación de tareas

El scheduler guarda varios datos sobre el estado de las tareas: por un lado, tiene un arreglo con el estado de las 16 tareas (en ejecución o detenidas); por el otro, mantiene en particular las últimas tareas de cada jugador, para poder asignarles tiempo en ronda.

También guarda la última tarea ejecutada para mantener este registro correctamente y es usada por otros servicios del sistema. Este valor puede no coincidir con la tarea indicada en el `tr`, ya que la tarea es desalojada al moverse, pero debemos respetar el orden de ejecución de todos modos.

Para identificar la siguiente tarea a ejecutar, el scheduler busca una tarea activa sobre el arreglo de tareas la siguiente tarea activa:

```
unsigned int sched_buscar_tarea(unsigned int jugador, unsigned int status) {
    unsigned int tarea = tareasAnteriores[jugador] + 1;
    if(tarea >= (TASK_PER_PLAYER * (1 + jugador))) {
        tarea = TASK_PER_PLAYER * jugador;
    }
    int i;
    for (i = 0; i < TASK_PER_PLAYER && status_tareas[tarea] != status; ++i) {
        ++tarea;
        if(tarea >= (TASK_PER_PLAYER * (1 + jugador))) {
            tarea = TASK_PER_PLAYER * jugador;
        }
    }
    return tarea;
}
```

Si la tarea retornada por este método es igual a la anterior, significa que el jugador posee como máximo 1 tarea activa.

Si la tarea retornada está inactiva, el jugador no tiene tareas activas, por lo que se busca una tarea del jugador actual. Si el mismo tampoco tiene tareas, o tiene una única tarea (y la misma coincide con el `tr`, es decir, no fue desalojada), no hace falta conmutar

Esta lógica también es utilizada para buscar una tarea libre en caso que deba lanzarse un zombi nuevo, por lo que no se asigna el primer zombi libre, sino que se asignan en secuencia. Esto podría modificarse tomando la tarea anterior por parámetro en lugar de buscarla en el arreglo dentro de este método.

El scheduler también provee la siguiente funcionalidad:

1. Una función para lanzar una tarea, que retorna su número asignado y la marca como iniciada

   ```
   unsigned int sched_lanzar_tarea(unsigned int jugador) {
       unsigned int nuevaTarea = sched_buscar_tarea(jugador, FALSE);
       status_tareas[nuevaTarea] = TRUE;
       return nuevaTarea;
   }
   ```

2. Una función para terminar la tarea en ejecución, que marca la misma como finalizada y salta a la tarea Idle (utilizando un jump far)

   ```
   void sched_matar_tarea_actual() {
       status_tareas[tareaActual] = FALSE;
       __asm __volatile("ljmp $0x68, $0" : : );
   }
   ```

3. Un toggle para el modo debug, ya que no se debe conmutar tareas durante el mismo

7.3. Servicio mover

Las tareas cuentan con una única manera de comunicarse con el kernel, esto lo logran realizando una interrupción a `0x66` indicando en *eax* hacia donde quiere moverse la tarea.

Para lograr esto dentro de la rutina de atención a la interrupción realizamos las siguientes acciones:

1. Pintar el rastro del zombi, es decir el lugar donde esta actualmente con un *(*)*.

   ```
   game_print_rastro(zombis[tarea].xPos, zombis[tarea].yPos);
   ```

2. Chequear si las condiciones para anotar un punto estan dadas. Si esto es asi se procede a matar a la tarea siguiendo los pasos ya indicados en el ejercicio 2.

   ```
   if(newXPos == 0 || newXPos == 79) {
       unsigned int puntoPara = newXPos == 0 ? JUG_B : JUG_A;
       if(++jugadores[puntoPara].score == 10) {
           game_finalizar();
       }
       game_print_score(puntoPara);
       game_matar_zombi_actual();
       sched_matar_tarea_actual();
   }
   ```

3. Caso contrario mapear y desmapear las paginas del zombi. Primero desmapear el area actual y luego mapear las nuevas paginas.

   ```
   mmu_mover_zombi(owner, zombis[tarea].xPos-1, zombis[tarea].yPos-1, newXPos-1, newYPos-1);
   ```

4. Sabiendo que tarea se esta ejecutando actualmente podemos saber que zombi es y a quien pertenece. Ademas sabemos hacia donde se va a mover chequeando el registro *eax*. Utilizar dicha información para dibujar la nueva ubicacion del zombie.

   ```
   game_print_zombi_mapa(tarea);
   ```

5. Saltar a idle.

   ```
   mov [sched_tarea_selector], word 0x68
   jmp far [sched_tarea_offset]
   ```

Luego de que termine el ciclo de clock en idle el scheduler se encargara de devolver la próxima tarea a ejecutar.

7.4. Modo debug

Si el modo debug se encuentra activado (mediante la tecla Y) el juego pasará a mostrar la siguiente excepción que se produzca en pantalla. Para lograr dicho cometido en la sección de manejo de excepciones se guarda de la información de todos los registros de uso común, los segmentos, la excepción que se disparo. Solo cuando el modo debug esta activado entonces esta información es pasada al juego el cual la almacena para poder usarla en un paso posterior.

Guardamos la información del mapa y mostramos la información de la excepción en un paso intermedio entre indicarle al scheduler que la tarea no se encuentra mas activa y saltar a idle (los pasos que se realizan al eliminar una tarea se encuentran en el ejercicio 2).

Salir del modo debug:

```
_isr33:

    in al, 0x60
    cmp al, key_debug
    je .toggle_debug

    test byte [debug_flag], debug_shown
    jnz .keyboard_end

    ...

    .toggle_debug:
        mov al, [debug_flag]
        test al, (debug_shown | debug_on)
```

```
        jz .enable_debug
        test al, debug_shown
        jz .keyboard_end

        ; disable_debug
        mov byte [debug_flag], debug_off
        call sched_toggle_debug ; indicar al scheduler que debe conmutar tareas nuevamente
        call game_debug_close ; restaurar el estado del mapa pre-debug
        jmp .keyboard_end

        .enable_debug:
        mov byte [debug_flag], debug_on
        jmp .keyboard_end
...
```

Para poder salir del modo debug y continuar el juego debemos presionar nuevamente la tecla *Y*, cuando lo hacemos dibujamos nuevamente la pantalla con la información que estaba guardada y el scheduler buscara la próxima tarea a ejecutar.