DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Organización del computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Otero, Fernando Gabriel	424/11	fergabot@gmail.com
More Ángel	931/12	angel_21_fer@hotmail.com
Gomez Arco Claudio	312/13	claudio4158@hotmail.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) 4576-3359
http://www.fcen.uba.ar

Índice

1. Ejercicio 1

A)

Utilizando el struct str_gdt_entry dada en $gdt.h$ completamos los primeros 8 (contando desde 0) descriptores con todos los campos en 0 ya que estos son descriptores nulos.
En la posición 9 está el descriptor de código de nivel 0. Este tiene que direccionar los primeros 624MB, pero como usamos granularidad tenemos que pasar la unidad de B a 4KB. Entonces la base es el resultado de esto (0x26EFF) y la acomodamos en las partes donde corresponde (porque está ”partida”). La base es 0. El tipo es A (ejecución y lectura). S (Descriptor type) es 1 porque es de tipo código. Dpl es 0 porque es código de nivel 0. D/b es 1 porque es para 32 bits. Presente está en 1 porque lo vamos a usar.
En la posición 10 está lo mismo que en la posición 9, pero dpl cambia a 3 porque es código de nivel 3.
En la posición 11 está el descriptor de datos de nivel 0, es lo mismo que en la posición 9 pero cambiando tipo por 2 ya que es de lectura/escritura.
En la posición 12 está lo mismo que en la 11 pero cambia el dpl a 3 ya que el descriptor de datos es de nivel 3. En la posición 13 está el descriptor de video, lo mismo que en la posición 12, cambiando la base por 0xB8000 y el dpl es 0.

Luego para entender mejor que descriptor es cada posición realizamos defines, en el archivo $defines.h$, para cada uno de ellos, así también sus offsets (posición * 8).

B)

En $kernel.asm$, habilitamos el pin A20 con call habilitar_A20. Cargamos la gdt con LGDT y GDT_DESC(función en $gdt.c$) ,luego seteamos el bit PE de CR0, e hicimos un JUMP para pasar a modo protegido en el segmento de código nivel 0 (el jmp es necesario para cambiar el valor del registro cs).
Luego, inicializamos los selectores de segmentos, data (ds) con el offset de la posición 10 (datos de nivel 0), así también para todos los segmentos restantes menos para fs que tiene el offset de la posición 12(desciptor de video). Por último pusimos en esp, y en ebp 0x27000 (ya que la pila debe comenzar en esta dirección).

C)

Para describir el área de la pantalla en memoria, para ser utilizado por el Kernel, se declaro la entrada de la posición 12 en la GDT, el mismo corresponde al segmento de video (Explicado en A) y el segmento correspondiente para video sera el fs, inicializado con el offset de dicha entrada.

D)

Se procedio a limpiar la pantalla con el coloreo básico, que es pintar todo el mapa de verde y las barras laterales de rojo y azul. Usando el segmento definido anteriormente. El pseudocódigo para realizar esto es el siguiente:

```
    ; la pantalla esta representada por una mariz de tamaño 50x80
    ; donde cada elemento ocupa 2 bytes
        xor ecx, ecx
        xor edx, edx
inc edx
xor esi, esi
add esi, 159
.ciclo: ;pinto de verde
cmp ecx, 8000 ;tamano de la pantalla
je .ciclo2
mov byte [fs:ecx], 0x22 ;verde en el fondo
inc ecx ;avanzo en la matriz
jmp .ciclo
.ciclo2: ; pinto barras
cmp edx, 8000
jae .mapa_ok
mov byte [fs:edx], 0x44 ;rojo
mov byte [fs:esi], 0x11 ;azul
add edx, 160
add esi, 160
jmp .ciclo2
.mapa_ok:
```

2. Ejercicio 2

A)

Utilizamos la macro dada en isr.asm, que dado un entero crea un código _isr(). Esta función llamará a la funcion en C $mostrarError$, luego desalojará la tarea actual y por ultimo pasará a la tarea idle, si esta no es la tarea actual.

```
%macro ISR 1
global _isr%1

_isr%1:
    mov EAX, %1
    PUSH EAX
    CALL mostrarError
    pop eax
.fin:
    call desalojar_tarea

    mov dword [estaIdle], 0x1
    mov dword [selector], 0x70
jmp far [offset]
    iret
%endmacro
```

MostrarError es un switch case, que dado un entero entre 0 y 20 mapea un error a un mensaje(descripción) y lo imprime por pantalla con la función print dada en screen.c, y por ultimo llama a la funcion $mDebugger()$. Luego utilizamos la macro ISR para i, $i = 0.,20$ En isr.h llamamos a _isr(i), para $i = 0.,20$, ya que esto será utilizado por idt.c, para obtener el offset de la rutina encargada de atender la interrupción.
Por último en idt.c inicializamos el arreglo de interrupciones con la función $idt\_inicializar$. Para esto, a su vez inicializamos a cada interrupción dentro del mismo por medio de una macro, $\#define\ IDT\_ENTRY(numero)$. La misma se encarga de completar todos los campos que poseen los descriptores de segmento de las interrupciones. Un offset que es la dirección de la función _isr correspondiente; segsel que se corresponde al segmento de código nivel cero. Y los atributos P=1; dpl correspondiente a la tarea y d= 1 (para 32bits).

B)

En kernel.asm llamamos a idt_inicializar(función en $idt.c$), que completa el vector de interupciones de las posiciones 0 a 19, 32, 33 y 102 con #define IDT_ENTRY(numero) de la forma descripta en el punto A.

La entrada 102 es la interrupción $mover$, la cual es llamada por las tareas. Por eso pusismos los atributos correspondientes (0xEE00) para que sea de nivel 3. Luego con LIDT cargamos el vector de interrupciones(idt_entry en $idt.h$) utilizando la estructura IDT_DESC definido en $idt.c$.

3. Ejercicio 3

A)

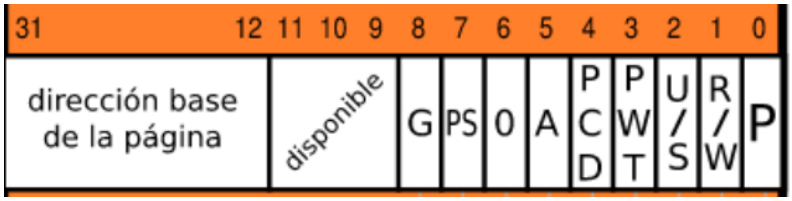
Para terminar de pintar la pantalla creamos una función en screen.c, iniciarPantalla. El pseudocódigo de la misma es el siguiente:

```
void iniciar_pantalla()
{
clockd[0] = '|';
clockd[1] = '/';
clockd[2] = '-';
clockd[3] = '\\';
Debugger = 0;

unsigned int x,y;
const char* texto = " ";
//Los espacios en negro:
for (x = 0; x < 80; x++)
{
print(texto, x, 0, 0x00);
print(texto, x, 45, 0x00);
print(texto, x, 46, 0x00);
print(texto, x, 47, 0x00);
print(texto, x, 48, 0x00);
print(texto, x, 49, 0x00);
}

for (x = 35; x < 45; x++)
{
for (y = 45; y < 50; y++)
{
if (x < 40)
{
print(texto, x, y, 0x44); //Franja roja (4 = rojo)
}
else
{
print(texto, x, y, 0x11); //Franja azul (1 = azul)
}
}
}
```

```
texto = "1 2 3 4 5 6 7 8";
print(texto, 5, 46, 0x0f);
print(texto, 60, 46, 0x0f);
print("( )", 75, 49, 0x0f);
texto = "# # # # # # # #"; //relojes
print(texto, 5, 48, 0x0f);
print(texto, 60, 48, 0x0f);

print("G", 0, 1, 0x44);
print("G", 79, 1, 0x11);

puntaje_o_restantes(juego.puntaje_B , 41, 47, 0x1f);
puntaje_o_restantes(juego._20A, 31, 47, 0x4f);
puntaje_o_restantes(juego.puntaje_A , 36, 47, 0x4f);
puntaje_o_restantes(juego._20B, 49, 47, 0x1f);

}
```

B)

El procesador posee una unidad de manejo de memoria, MMU, (Memory Management Unit) que es un dispositivo de Hardware responsable del manejo de los accesos a memorias, entre algunas de sus funciones. Este dispositivo se va a encargar de la traducción de las direcciones lógicas (o virtuales) a direcciones físicas (o reales) que enviara por el bus de address hacia la memoria externa. Para representar esto se cuenta con un directorio de paginas, cada pagina tiene un tamaño de 4k y 1024 descriptores de 8 bytes. Cada uno representa en los bits 31 a 12 el valor correspondiente a la dirección física de la Pagina (el Page Frame) que contiene la tabla de descriptores de pagina, cada una de estas con 1024 descriptores. Además cada descriptor de 32 bits posee las siguientes características:



Para representar los descriptores y con esto el directorio de paginas se utilizo un str en lenguaje C, como el que se muestra a continuación y en el archivo mmu.c

```
typedef struct str_mmu_entry{
\textit{ unsigned char p:1;
unsigned char rw:1;
unsigned char us:1;
unsigned char pwt:1;
unsigned char pcd:1;
unsigned char a:1;
unsigned char ign:1;
unsigned char ps:1;
unsigned char g:1;
unsigned char disp:3;
unsigned int  base\_0\_20  :20; }
} \textit{__attribute__((__packed__, aligned (4))) mmu_entry};
```

estructura que representa un descriptor de directorio de pagina y de la page table. Si bien entre si difieren algunos de sus parámetros, los que lo hacen no se utilizan.

mmu_inicializar_dir_kernel()

En la page table cada descriptor, con el atributo P=1, contiene la dirección de una pagina correspondiente a una Tabla de Paginas. Este es el ultimo nivel de traducción. Cada pagina es de 4K, con 1024 descriptores de 4bytes cada uno. Y cada uno de estos posee en los bits 31 a 12, el Page Frame de la pagina de memoria, es decir los 20 bits mas significativos de la dirección física de memoria en donde comienza la pagina.
Como se desea crear un directorio de paginas que mapee usando identity mapping, las direcciones 0x00000000 a 0x3FFFFF, se necesita completar 1 pagina, de la page table ya que cada descriptor se corresponde con un pagina en dirección física de 4K y en total se estarían completando 1024 descriptores, como cada uno se corresponde con 4k obtenemos 1024*4 = 0x400000 que es lo que necesitamos direccionar. Como fue necesaria una pagina, se completó un descriptor de la page directory con los siguientes atributos (la inicialización de todos las paginas y descriptores mencionados se encuentran mmu.c). En primer lugar se declaro que la dirección inicial del directorio de pagina es la 0x27000: *mmu_entry *pd = (mmu_entry *) 0x27000; p = 1; rw = 1; us = 0; pwt = 0; pcd = 0; a = 0; ign = 0; ps = 0; g = 0; disp = 0;* debido a que los 20 bits mas significativos constituyen la dirección del directorio de pagina, se shiftea 12 bits hacia la izquierda para quedarnos con los restantes $base\_0\_20$ = (0x28000 + i*0x1000) $>>$ 12;
Como cada pagina posee 1024 descriptores, el resto de los descriptores sin usar fueron inicializados con los mismo atributos excepto que P=0 , ya que no se necesitan. Una vez realizado esto se inicializaron los ya mencionados descriptores de pagina. Como ya se cuenta con un directorio de pagina inicializado en la dirección 0x27000 y esta ocupa 0x1000, las tablas de pagina van a comenzar a partir de la dirección 0x28000. Y los atributos tendrán el valor *mmu_entry *pt = (mmu_entry *) 0x28000 ; p = 1; rw = 1; us = 0; pwt = 0; pcd = 0; a = 0; ign = 0; ps = 0; g = 0; disp = 0; base_page_0_20 = i*0x1000*; Donde 0 $<=$ i $<=$ 1024, ya que de esta manera se direcciona a los primero 4Mb de memoria.

C)

En $kernel.asm$ llamamos a mmu_inicializar lo que hace es empezar un contador de páginas libres. Llamamos a mmu_inicializar_dir_kernel que hace lo descripto arriba.
Colocamos en CR3 0x27000 porque es donde empieza la page directory y seteamos el primer bit de CR0, todo esto para habilitar paginación.

D)

Creamos un texto con el nombre del grupo de la misma forma que estaba hecho para $Iniciandokernel(ModoReal)$...(en $kernel.asm$), luego utilizando imprimir_texto_mp, imprimimos el texto que definimos. Para centrarlo a derecha a 80 le restamos la longitud del texto y este valor lo pusimos en y, como tiene que estar en la primera fila x es 0. El color que elegimos fue gris con fondo negro por eso pusimos 0x07 para el color.

4. Ejercicio 4

A)

4.2 Inicializando la MMU:

Para inicializar se completo la función $inicializar\_mmu$ en el archivo mmu.c. Como varias funciones utilizan un pedido de pagina libre, se declaro una variable global que es la encarga de dar a las mismas. $inicializar\_mmu$ va a ser la encargada de asignar el área con el que va a comenzar dicha variable. Finalmente para que estas funciones se pongan en funcionamiento se las llamo desde kernel.asm. Tanto a $mmu\_inicializar$ como a $mmu\_inicializar\_dir\_kernel()$.

B)

mmu_inicializar_dir_zombie():

Esta función se va a encargar de inicializar un directorio y tablas de página para una tarea determinada. Se cuenta con un total de 8 tareas por jugador. Por cada clase de zombie y jugador se posee una tarea, y una página de código que debe ser copiadas dentro de el_mapa es decir, a partir de la dirección 0x400000. Y también se disponen de 9 paginas que deben ser mapeadas al mapa, las mismas se encuentran en la dirección virtual 0x8000000.
Para estos últimos mapeos se cuenta con la función mmu_mapear_pagina (descripta en el siguiente item). Como son 9 paginas por jugador, se cuenta con mapeoAlrededor. La misma calcula la posición en donde deben ser mapeadas las paginas y las mapeas utilizando la función mmu_mapear_pagina. Como las paginas se utilizaran para las tareas y las mismas corren en nivel de usuario attr sera igual a 3. Y como por cada zombie hay que construir un mapa. Cada uno va a poseer un cr3 distinto. El mismo se consigue pidiendo una pagina libre del área libre. No solo es necesario mapear estas paginas sino que para no perder información e inicializar la page directory y la correspondiente page table se deben mapear los primeros 4 Mb y este sector se mapea con identity mapping. Entonces la función creardtpt, que recibe el cr3, se va a encargar de realizar esta tarea. Esta función realiza lo mismo que $mmu\_inicializar\_dir\_kernel()$, solo que la base de la page directoy ahora es la dirección del nuevo Cr3. Por ultimo, como se menciono anteriormente, es necesario copiar el código de la tarea al mapa. Para esto se va a contar con la función copiar_código_zombie. Donde dado un jugador y la dirección del mapa se va a encargar de copiar el código que corresponda al área del mapa.

c)

mmu_mapear_pagina: dada una dirección física, un cr3, una dirección, virtual y el nivel de privilegio de lo que se desea mapear, attr. Mapea de virtual a física, su funcionamiento es muy similar a los descripto en $mmu\_inicializar\_dir\_kernel()$, solo que ahora la base de la page directory es un cr3 pedido al área libre. Y el párametro $base\_0\_20$ de la page directory va a ser referencia a la dirección cr3 + 0x1000 , que es donde se va a inicializar su page table. Como ahora se desea mapear a una dirección física determinada, el campo $base\_0\_20$, de las entradas de las page table, debe apuntar a la física deseada. Y se llama a la función tlbflush para que se invalide la cache de traducción de direcciones.
Finalmente, como cada vez que un zombie se mueve es necesario desmapear las paginas donde se encontraba anteriormente, se cuenta con la función $mmu\_unmapear\_pagina$. Que dado una dirección virtual y un cr3. Se va a encargar de setear el bit de presente con el valor cero de la entrada de la page table apuntada por la dirección virtual. Para lograr esto, primero es necesario saber la dirección de la page directory, que se consigue con el cr3 dado. Para saber cual es la entrada de la page directory nos quedamos con los primeros 10 bits de la dirección virtual. Obtenida, con el campo $base$ de esta entrada y los siguientes 10 bits de la virtual. Nos ubicamos en la page table que queremos y cambiamos el valor de p a cero.

5. Ejercicio 5

A)

En el archivo idt.c se definió a la función $idt\_inicializar$ (descripta en el punto 2). En esta instancia es necesario definir tres nuevas entradas: $IDT\_ENTRY(32)$; $IDT\_ENTRY(33)$; $IDT\_ENTRY(102)$; La primera se utiliza para asociar una rutina a la interrupción del reloj, la segunda para la interrupción del teclado y la última para la interrupción del software 0x66. Como se están agregando tres nuevos descriptores es necesario completar sus campos. Por lo que en el archivo isr.h se definieron las funciones $\_isr32()$, $\_isr33()$ e $\_isr102()$; Así podemos obtener el offset de cada una. Y finalmente se definió en el archivo irs.asm una rutina para cada una, encargada de validar dichas interrupciones y que realicen lo deseado.

Primera interrupción, Reloj:
como La maquina posee un reloj interno que genera interrupciones en ciertos intervalos regulares de tiempo. Se utilizaran estas interrupciones para que se ejecute una rutina cada vez que esto sucede. Principalmente se completaran los atributos de su descriptor con P = 1(segmento presente); DPL= 0 (nivel de privilegio), y el offset con la dirección de la función $\_isr32()$.
Segunda interrupción, Teclado:
Los campos del descriptor se completaron con los mismos valores que para la anterior interrupción, variando el offset que ahora es la de la función $\_isr33()$.
Tercera Interrupción, 0x66:
A diferencia de las otras interrupciones, esta es provocada por las tareas como la misma se ejecutan a nivel de usuario (3). El dpl de este descriptor debe ser 3. mientras que los demás atributos tienen el mismo valor que para las anteriores interrupciones, con el offset correspondiente (isr_102).

B)

Primera interrupción, Reloj:
En $isr.asm$ el código se encarga de:
Deshabilitar la interrupciones (cli) preservando los registros que hay hasta el momento ($popad$). Se llama a la función $proximo\_reloj$ para que cada vez que se produzca un tick la función se encargue de mostrar en el margen inferior de la pantalla una animación de un cursor rotando. Por ultimo se comunica al PIC que se atendió la interrupción (fin_intr_pic1), se restauran los registros ($popad$), se vuelven a activar las interrupciones (sti) y se retorna de las interrupciones ($iret$). (posteriormente esta función se modificó para adaptarse a lo pedido en el punto 7, más adelante se describirá esta modificación).

C)

Segunda interrupción, Teclado:
En isr.asm también se procedió a deshabilitar las interrupciones y preservar los registros. Como el teclado se lee a través del puerto 0x60, por medio de la instrucción in al, 0x60. Cada vez que se presiona una tecla se consigue un scan code de la misma. Como según la tecla oprimida el juego realiza distintas acciones. Se crea una función, en $screen.c$, llamada teclado. La misma recibe por parámetro el scan code de la tecla oprimida. Si el mismo, no es el correspondiente a ninguna de las teclas del juego entonces no se realiza nada. De lo contrario, se cuenta con una struct que representa al juego, str_game (definida en $game.h$). La misma cuenta con atributos como filaA, filaB, claseA, claseB; Los primeros dos representan la fila desde donde se lanzara al Zombie según el jugador. Y los últimos dos la clase del zombie lanzado. Por lo que si el scan code recibido corresponde a las tecla W o S, se incrementa o se reduce el valor de filaA, correspondientemente. Y lo mismo sucede con claseA cuando se oprima A o D. Análogamente, con las teclas correspondiente, se modifica filaB o ClaseB. Además si se oprime shiftLeft, se llama a la función $game\_lanzar\_zombie$ (función descripta mas adelante) con el parámetro cero, el mismo hace referencia al Jugador A. En el caso de oprimirse ShiftRight se llama a la misma función pero con 1, jugador

B . Finalizada las acciones correspondientes se restauran los registros, se comunica que se atendió la interrupción y se vuelve de la misma.

D)

Tercera Interrupción, 0x66:
Al igual que en las anteriores, en $isr.asm$, se comienza deshabilitando las interrupciones. Se preservan los registros, y se procede a realizar lo pedido en esta instancia. Solo se debe modificar el valor de un registro (eax). Finalizado, se comunica que se atendió la interrupción, se restauran los registros y se vuelve de la interrupción. (Posteriormente esta interrupción sera modificada para adaptarse a lo pedido en el ejercicio 7)

6. Ejercicio 6

A)

Dado que GDT_COUNT(en defines.h) estaba en 30 y no nos alcanzaba lo cambiamos a 31.
En la posición 13 de la GDT(tarea_inicial) todos los campos van en 0,menos limite que es el tamaño de una tss -1, tipo es 9 porque es sólo ejecución y acceso, s va en 0 porque es del sistema, dpl es 0 porque es del kernel, p es 1 porque está presente, d/b es 1 porque es a 32 bits y g está en 1 porque hay granularidad.
En la posición 14(idle) tenemos lo mismo que en la 13.
De la posición 15 a la 22 inclusive tenemos los descriptores de la tss de las 8 tareas del jugador A, todo se completa igual que la posición 13. De la posición 23 a la 30 inclusive tenemos los descriptores de la tss de las 8 tareas del jugador B. En estos casos como esta completada tiene poca importancia porque al ser creado el zombie se reescriben los campos de la tss correspondiente, para el caso que se rehusen con un nuevo zombie (donde hay que poner el estado inicial de la tarea), como es explicado en la sub-sección C Algunas cosas serán completadas más adelante ,como la base, ya que obtendremos la dirección de las tss cuando sea necesarios crearlas.

B)

En el archivo tss.h armamos una struct str_tss para completar las tss de las tareas de la siguiente forma:

```
typedef struct str_tss {
    \textit{unsigned short  ptl;
    unsigned short  unused0;
    unsigned int    esp0;
    unsigned short  ss0;
    unsigned short  unused1;
    unsigned int    esp1;
    unsigned short  ss1;
    unsigned short  unused2;
    unsigned int    esp2;
    unsigned short  ss2;
    unsigned short  unused3;
    unsigned int    cr3;
    unsigned int    eip;
    unsigned int    eflags;
    unsigned int    eax;
    unsigned int    ecx;
    unsigned int    edx;
    unsigned int    ebx;
    unsigned int    esp;
    unsigned int    ebp;
    unsigned int    esi;
    unsigned int    edi;
    unsigned short  es;
    unsigned short  unused4;
    unsigned short  cs;
    unsigned short  unused5;
    unsigned short  ss;
    unsigned short  unused6;
    unsigned short  ds;
    unsigned short  unused7;
    unsigned short  fs;
    unsigned short  unused8;
    unsigned short  gs;
    unsigned short  unused9;
    unsigned short  ldt;
    unsigned short  unused10;
    unsigned short  dtrap;
    unsigned short  iomap;]}
} \textit{__attribute__((__packed__, aligned (8))) tss;}
```

Acá es donde completamos la base de la tarea idle en la GDT, pedimos la dirección de tss_idle(de la str anterior creada para ella) y shifteando ”partimos”la base para ubicarla en las partes $base$ de la GDT en la posición de la tarea idle.
Luego para tss_idle, ponemos todos los campos en 0 menos, $cr3$ que tiene el mismo que el kernel y es 0x27000, eip en 0x16000 que es donde se encuentra la tarea, $eflags$ 0x0202 para que se activen las interrupciones, esp y ebp en 0x27000 que es la misma que la del kernel, en cs ponemos la posición del desciptor de código nivel 0 en la GDT multiplicado por 8 (porque es el offfset), en es, ss, ds, fs y gs ponemos la posición del descriptor de datos de nivel 0 en la GDT multiplicado por 8 (porque es el offset), y por último $iomap$ tiene 0xffff que es algo que no tocamos.

C)

Tenemos 2 vectores de tss(en el archivo $tss.c$), uno para cada equipo con 8 tss dentro. Además contamos con 2 funciones $tss_inicializar_tarea_zombieA$ y $tss_inicializar_tarea_zombieB$ ambas hacen lo mismo pero se diferencian en el nombre para resumir las tomas de decisiones según que jugador es el que lanza el zombie. Esta función toma un id (el número de zombie) y el $CR3$ correspondiente (las páginas correspondientes a este zombie ya fueron mapeadas). Primero al igual que la idle completa la base en la GDT de la tarea id. Para ello se toma id y se le suma un offset, que es donde empiezan las tareas del equipo que forma parte(si es A es tarea A0=15 y si es B tarea es B0=23), para saber en que posición de la GDT está esa tarea que se quiere inicializar. Sabiendo la entrada de la GDT correspondiente, partimos la posición de la TSS para ubicarla en el campo $base$ de la GDT.
La tss del zombie la vamos a completar con todos los campos en 0 menos $esp0$ que tiene una posición de memoria libre + 0x1000, $ss0$ tiene el offset del segmento de la gdt de datos nivel 0, $eflags$ = 0x202, esp y ebp tienen 0x08000000+0x1000, es, ss, ds, fs y gs tienen el offset del segmento de datos nivel 3 + 3 (por que tiene nivel de priviligio 3), cs tienen el offset del segmento de codigo nivel 3 + 3, y por último $iomap$ tiene 0xFFFF.

D)

Para la tss de la $tareainicial$ ponemos la base de la tarea incial en la GDT ,de la misma forma que la idle, pedimos la dirección de tss_inicial y shifteando ”partimos”la base para ubicarla en las partes $base$ de la GDT en la posición de la tarea inicial.
No necesitamos completar nada de los campos porque al saltar de la tarea inicial a la siguiente, ésta TSS no se lee sinó que se escribe.

E)

Explicado en A.

F)

Para realizar lo pedido escribimos en $kernel.asm$ lo siguiente:

```
;Cargar tarea inicial
mov ax, 0x68
ltr ax

;Saltar a la primera tarea: Idle
;jmp 0x70:0 ; 14 * 8 (posicion de la idle en la gdt)
mov dword [selector], 0x70
jmp far [offset]}
```

7. Ejercicio 7

A)

El scheduler fue armado a través de varias partes, en particular la isr32 que cicla las tareas. La funcion sched_proximo_indice (sched.c) se encarga de devolver el selector de la siguiente tarea para que sea ejecutada. Los elementos necesarios para estas funciones, que deben ser inicializados, están en el struct str_game juego (game.h) y son inicializados junto a otras variables globales en game.c.

```
typedef struct str_game_tp {
tarea tareasA[9]; //.p en 0 si estan libres, en 1 si estan ocpuadas (son 9 para facilitar
//la opcion "todas ocupadas")
tarea tareasB[9];
unsigned int tareaActualA; //sera la ultima (o actual) tarea del jugador
unsigned int tareaActualB;
unsigned int tareaRemovida; //de 0 a 15, 0-7 = A y 8-15 = B (para el debugger)
unsigned int filaA; //las filas son donde se encuentra el jugador
unsigned int filaB;
int claseA; //la clase que tiene seleccionada el jugador
int claseB;
unsigned int jugadorActual; //para saber que jugador esta en el tic de reloj actual
int zombis_restantes_a; // 8 menos esto es la cantidad que se pueden lanzar.
int zombis_restantes_b;
int puntaje_A;
int puntaje_B;
unsigned int debugger; //flag debugger
unsigned int _20A; //zombies que le quedan por lanzar
unsigned int _20B;
} __attribute__((__packed__)) str_game;
```

```
void inicializarJuego()
{
estaIdle = 1;
tiempo_esperando = 0;
juego.zombis_restantes_a = 8;
juego.zombis_restantes_b = 8;
juego._20A = 20;
juego._20B = 20;
juego.puntaje_A = 0;
juego.puntaje_B = 0;

puntaje_o_restantes(juego._20A, 31, 47, 0x4f);
puntaje_o_restantes(juego._20B, 49, 47, 0x1f);

puntaje_o_restantes(juego.puntaje_A , 36, 47, 0x4f);
puntaje_o_restantes(juego.puntaje_B , 41, 47, 0x1f);

int i = 0;
for (i = 0; i < 9; i++)
{
juego.tareasA[i].p = 0;
juego.tareasB[i].p = 0;
}
juego.tareaActualA = 0;
juego.tareaActualB = 0;
juego.tareaRemovida = 16; //0-7 = A, 8-15 = B, 16 invalido (no se removio ninguna tarea)
juego.filaA = 1;
juego.filaB = 1;
juego.claseA = 0;
juego.claseB = 0;
juego.jugadorActual = 0;
juego.debugger = 0;
}
```

B)

sched_proximo_indice devolverá el selector de la siguiente tarea a ejecutar. Tiene en cuenta las distintas combinaciones de cantidades de zombies y trabaja con las variables globales de **juego**. Lo que hace sched_proximo_indice es revisar a quien le toca jugar (en la variable juego.jugadorActual se encuentra el último en jugar, asi que debe tomar el contrario), y de ése jugador revisará cual es el zombie para lanzar. En caso de que no haya ninguno, lanzará un zombie del jugador contrario, y en caso de que ninguno tenga zombies devolverá 0 para que no salte a ninguna tarea (porque siempre la anterior tarea será la Idle). Si el jugador del turno actual no tiene zombies y se lanza un zombie del jugador contrario, se avisará por medio de la variable juego.jugadorActual este hecho para que no se salteen turnos en ningún caso.

```
if (juego.jugadorActual == 0)
{ //si el actual es A, ahora pasa a jugar B
res = sched_indice_B();
if (res == 8) { //si no hay zombies de B...
res = sched_indice_A();
if (res == 8) { //si no hay zombies de A..
res = 0;
} else { //si hay zombies de A..
if (juego.tareaActualA == res && !estaIdle) {
//si hay un solo zombie de A..
res = 0;
} else {
juego.tareaActualA = res;
res = (res + GDT_IDX_TSS_A0) *8;
}

juego.jugadorActual = 1;
}
} else { //si hay zombies de B..
juego.tareaActualB = res;
res = (res + GDT_IDX_TSS_B0) *8;
}
}
```

C)

Para este punto se modificó el funcionamiento de la rutina de atencion de int 0x66 (isr102) que se tenia en el punto 5. Ahora, se va a encargar de mover al zombie de la tarea que se esta ejecutando. La interrupción llama a game_move_current_zombi. Una vez realizada la acción correspondiente se procede a saltar a la tarea idle, si no se esta ejecutando por alguna razón (como puede ser que tire una excepción).

isr102 recibirá por eax la dirección a la que se debe mover. game_move_current_zombi lo toma y calcula la nueva posición. Guarda la vieja posición para saber de donde copiar el código de la tarea zombie.

game_move_current_zombi toma en cuenta los casos de que el zombie llegue a la fila superior o inferior, ya que el mapa es cilindrico. También ve los casos en los que llega a uno u otro extremo, ya que debe desalojar la tarea (con .p = 0) y darle el punto al jugador correspondiente.

```
_isr102:
cli
pushad
push eax
call game_move_current_zombi
cmp BYTE [estaIdle], 0x1
je .fin
mov dword [estaIdle], 0x1
mov dword [selector], 0x70

jmp far [offset]
jmp .fin
.fin:
pop eax
popad
iret
```

D)

La atención de la interrupción del reloj saltará siempre que haya al menos un zombie (si no hay, sched_proximo_indice devolverá 0). cambiar_jug_actual modifica juego.jugadorActual. El cambio de tareas, que se realiza dentro de la isr32 tiene los casos de cambio de tarea a la idle, a un zombie o no saltar (siempre que se deba saltar a la tarea que se está corriendo en ese momento).

Ademas, si en un tiempo determinado (1000 ticks) ningún jugador realiza un lanzamiento (aún habiendo zombies por lanzar) y ningún zombie se mueve, el juego finaliza utilizandó como criterio de ganador el valor que se tenga hasta el momento de $puntaje\_A$ y $puntaje\_B$. Para esto, cada vez que se produzca un tick de reloj (fuera del modo debugger) se incrementá una variable global llamada $tiempo\_esperando$ (inicializada en cero). Pero, si un jugador lanza un zombie o se produce un movimiento de los mismos vuelve al valor cero. En caso de que se llegue al valor 1000 se llama a la función $ganador$, la misma verifica el valor de parámetro de entrada. Si es 1, que es este caso, se muestra por pantalla quien es el ganador y se setea el valor de $\_20A$ y $\_20B$ como cero, para indicar que ya no se pueden lanzar más zombies y dar por finalizado el juego. Si es otro valor y $\_20A$ y $\_20B$ son iguales a cero se lanzaron todos los zombies y se muestra al ganador. En cualquiera de los dos casos, luego de finalizar el juego se salta a la tarea idle.

Este método de parada fue diseñado por el hecho de que una tarea puede ser modificada por otra tarea en el medio de campo, y como consecuencia dejar de moverse.

```
_isr32:
cli
pushad
;prioridad: si estamos en el debugger no se toca nada
mov eax, 1
cmp eax, [Debugger]
je .debugger
;si no estamos en el debugger, y no se toco nada hace 1000
;ciclos de reloj, asumo que el juego termino.
inc dword[tiempo_esperando]
cmp dword [tiempo_esperando], 1000
jge .termino
;dibujo los clocks
call clock
call proximo_reloj
;paso a la tarea siguiente (o me quedo en esta si son la misma
call sched_proximo_indice
push eax
call cambiar_jug_actual
pop eax
cmp ax, 0
je .noJump
mov [selector], ax
mov dword [estaIdle], 0
call fin_intr_pic1
jmp far [offset]
jmp .end
.termino:
xor eax, eax
inc eax
push eax
call ganador
pop eax
.debugger:
cmp byte [estaIdle], 1
je .noJump
mov dword [estaIdle], 0x1
mov dword [selector], 0x70
jmp far [offset]
.noJump:
call fin_intr_pic1
.end:
popad
iret
iret
```

E)

El desalojo de las tareas al producirse una excepción se realiza en $desalojar\_tarea$, la cual desaloja simplmente poniendo 0 en la propiedad p de la tarea, en el struct $juego$. Además de anularla habilita al jugador de la tarea desalojada a lanzar un nuevo zombie.

```
%macro ISR 1
global _isr%1

_isr%1:
    mov EAX, %1
    PUSH EAX
    CALL mostrarError
    pop eax
.fin:
    call desalojar_tarea

    mov dword [estaIdle], 0x1
    mov dword [selector], 0x70
jmp far [offset]
    iret
%endmacro
```

F)

El mecanismo de debugging utilizará en primer lugar juego.tareaRemovida, variable que devolverá qué tarea es de la que tiene que mostrar información, y juego.debugger que servirá de flag para indicar si debe activarse el debugger o no. Esta variable se modifica al pulsar la tecla 'y'. La función principal que se encarga de controlar el proceso es mDebugger(), la cual se encarga de copiar el sector de la pantalla a una variable global para tal fin, y luego cargar el fondo y las etiquetas de cada registro. Los datos son cargados con cargoDebugger(), que lee los datos de la tss de la tarea recién desalojada. Luego, se queda en la tarea Idle hasta que una interrupción de teclado hace salir del bucle.

```
void mDebugger()
{
//guarda la pantalla, pega la pantalla y luego llama a una funcion en asm q imprime el estado de los registros
//guardo columnas 25-55, filas 6-42 (36 filas, 30 cols)
if (juego.debugger)
{
guardoPantalla();
Debugger = 1;

//[...] dibujo la pantalla del debugger

cargoDebugger();
}
}
```

```
void cargoDebugger()
{
unsigned int base2;
base2 = (unsigned int)gdt[juego.tareaRemovida + GDT_IDX_TSS_A0].base_0_15;
base2 += (unsigned int)gdt[juego.tareaRemovida + GDT_IDX_TSS_A0].base_23_16 << 16;
base2 += (unsigned int)gdt[juego.tareaRemovida + GDT_IDX_TSS_A0].base_31_24 << 24;

tss* base = (tss*)base2;
//base apunta a la tss que desaloje
if (juego.tareaRemovida < 8)
{
switch (juego.tareasA[juego.tareaActualA].clase)
{
case 0:
print("Zombie  A  Guerrero", 26, 7, 0x1f);
break;
case 1:
print("Zombie  A  Mago", 26, 7, 0x1f);
break;
case 2:
print("Zombie  A  Clerigo", 26, 7, 0x1f);
break;
}
}
else
{
switch (juego.tareasB[juego.tareaActualB].clase)
{
case 0:
print("Zombie  B  Guerrero", 26, 7, 0x1f);
break;
case 1:
print("Zombie  B  Mago", 26, 7, 0x1f);
break;
case 2:
print("Zombie  B  Clerigo", 26, 7, 0x1f);
break;
}
}

// [...] escribo en pantalla el valor de los registros pedidos y la pila de la tarea
}
```

Es importante notar que luego de cargarse el debugger, al estar la variable Debugger en 1, la interrupción isr32 (reloj) no cambiará de tareas y se quedará en la idle (o cambiará a esta de encontrarse en otra tarea) hasta que se produzca una interrupción de teclado. En la función **teclado**, si la interrupción fue dada por precionar una tecla, actualizará el valor de la variable Debugger en 0, haciendo que se cargue el sector de la pantalla que fue removido al pasar al modo debugger.