



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System programming

Organización del Computador II
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Leandro Raffo	945/12	leandrojavr@gmail.com
Maximiliano Fernández Wortman	892/10	maxifwortman@gmail.com
Uriel Rozenberg	838/12	urielrozenberg@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.	3
2. Ejercicio 2.	4
3. Ejercicio 3.	5
4. Ejercicio 4.	8
5. Ejercicio 5.	10
6. Ejercicio 6.	13
7. Ejercicio 7.	19
7.1. a)	19
7.2. b)	19
7.3. c) y e)	20
7.4. d)	20

1. Ejercicio 1.

Básicamente armamos la tabla de gdt, empezando en la posición 8 del array, con 4 segmentos, dos de código nivel 0 y 3 y dos de datos nivel 0 y nivel 3, llamados respectivamente:

```
GDT_IDX_CS_CERO_DESC  
GDT_IDX_CS_TRES_DESC  
GDT_IDX_DS_CERO_DESC  
GDT_IDX_DS_TRES_DESC
```

Excepto el dpl y el tipo, todos estos segmentos tienen un límite de 0x1E400, con base en 0x0000, ya que esto nos da 0x1F400 que son 500mb de memoria, esto es porque tenemos la granularidad en 1 es decir contamos de a 4k.

Para pasar a modo protegido usamos el siguiente código.

```
    jmp 0x40:modoprotegido  
BITS 32  
    modoprotegido:  
    mov eax, 0x50  
    mov ss, ax  
    mov ds, ax  
    mov gs, ax  
    mov fs, ax  
    mov es, ax  
    mov ebp, 0x1337  
    mov esp, 0x28000
```

Hacemos un far jump con el offset para CS tenga el valor de nuestro código en nivel 0 que es $8 \ll 3 = 0x40$ y ponemos el comando BITS 32 para que el ensamblador sepa que las próximas instrucciones sean generadas para un procesador de 32 bits, seteamos todos los registros de segmentos con el de datos de nivel 0 ($10 \ll 3 = 0x50$) y definimos el stack esp en 0x28000 como pide el enunciado.

Para el punto siguiente usamos la función call screen_inicializar para pintar la pantalla.

2. Ejercicio 2.

Tenemos que definir las interrupciones, para esto usamos el macro `IDT_ENTRY`, este se encuentra en `idt.c` y lo llamamos con los siguientes valores adentro de la función `idt_inicializar`, donde el primer valor es el número de la interrupción y el segundo el del dpl:

```
IDT_ENTRY(0, 0);    /* Divide error          */
IDT_ENTRY(2, 0);    /* NMI Interrupt          */
IDT_ENTRY(3, 0);    /* Breakpoint - INT 3     */
IDT_ENTRY(4, 0);    /* Overflow               */
IDT_ENTRY(5, 0);    /* BOUND Range Exceeded   */
IDT_ENTRY(6, 0);    /* Invalid opcode         */
IDT_ENTRY(7, 0);    /* Device Not Available    */
IDT_ENTRY(8, 0);    /* Double Fault           */
IDT_ENTRY(9, 0);    /* Coprocessor segment overrun */
IDT_ENTRY(10, 0);   /* Invalid TSS            */
IDT_ENTRY(11, 0);   /* Segment not present    */
IDT_ENTRY(12, 0);   /* Stack-Segment Fault    */
IDT_ENTRY(13, 0);   /* General Protection     */
IDT_ENTRY(14, 0);   /* Page Fault             */
IDT_ENTRY(16, 0);   /* x87 FPU Floating-Point Error */
```

```
IDT_ENTRY(17, 0);   /* Alignment Check        */
IDT_ENTRY(18, 0);   /* Machine Check           */
IDT_ENTRY(19, 0);   /* SIMD Floating-Point Exception */
```

Una vez hecho esto en `isr.asm` usamos el macro

```
%macro ISR 1
global _isr%1

_isr%1:
    mov eax, %1
    jmp $

%endmacro
```

Para definir las 18 interrupciones.

```
ISR 0
.
.
.
ISR 19
```

Luego de esto en `kernel.asm` pusimos el siguiente código:

```
call idt_inicializar
lidt [IDT_DESC]      ; igual que con la gdt
```

Es decir inicializamos la `idt` y la cargamos a su registro correspondiente. Para probar que esto andaba hicimos una división por zero:

```
mov eax, 0
mov ebx, 0
div ebx
```

Lo cual hizo saltar la excepción 0, de divide error.

3. Ejercicio 3.

a), b), c)

Para este punto programamos la función `mmu_mapear_pagina`:

```
void mmu_mapear_pagina(uint virtual, uint cr3, uint fisica, uint attrs) {
    cr3 &= 0xFFFFF000;
    uint directorio = (virtual >> 22) & 0x3FF;    /* Posicion en el directorio */
    uint tabla = (virtual >> 12) & 0x3FF;        /* Posicion en la tabla */
    page_directory *pd = (page_directory *) cr3;

    if (pd[directorio].present == NULL) {
        uint proxima_pag = mmu_proxima_pagina_fisica_libre();
        mmu_inicializar_page_directory(&pd[directorio], proxima_pag, 0x3);
        int tab_c = proxima_pag;
        for (; tab_c < proxima_pag + 0x1000; tab_c += 0x4)
            mmu_inicializar_page_table((page_table *)tab_c, 0, 0);
    }

    page_table *pt = (page_table *) (pd[directorio].base_adress << 12);
    mmu_inicializar_page_table(&pt[tabla], fisica, attrs);
}
```

Esta función recibe una dirección virtual, un `cr3`, una dirección física y los atributos. Primero conseguimos la dirección del directorio haciendo un and a `cr3` con `0xFFFFF000` y lo apuntamos por `pd` un puntero a un directorio, `page_directory *`, donde `page_directory` es el siguiente struct:

```
typedef struct page_directory {
    unsigned char    present:1;
    unsigned char    read_write:1;
    unsigned char    user_supervisor:1;
    unsigned char    page_level_write_through:1;
    unsigned char    page_level_cache_disabled:1;
    unsigned char    accessed:1;
    unsigned char    ignored:1;
    unsigned char    page_size:1;
    unsigned char    global:1;
    unsigned char    available_11_9:3;
    unsigned int     base_adress:20;
} __attribute__((__packed__, aligned (4))) page_directory;
```

Luego nos fijamos si está presente, de no estarlo le pedimos una página física libre a `mmu_proxima_fisica_libre` y la inicializamos con la función `mmu_inicializar_page_directory`:

```
void mmu_inicializar_page_directory(page_directory * dir, uint addr, uint attrs) {
    dir->present = attrs & 0x1;
    dir->read_write = (attrs >> 1) & 0x1;
    dir->user_supervisor = (attrs >> 2) & 0x1;
    dir->page_level_write_through = (attrs >> 3) & 0x1;
    dir->page_level_cache_disabled = (attrs >> 4) & 0x1;
    dir->accessed = (attrs >> 5) & 0x1;
    dir->ignored = (attrs >> 6) & 0x1;
    dir->page_size = (attrs >> 7) & 0x1;
    dir->global = (attrs >> 8) & 0x1;
    dir->available_11_9 = (attrs >> 9) & 0x3;
    dir->base_adress = addr >> 12;
}
```

Luego usando la función `mmu_inicializar_page_table`

```
void mmu_inicializar_page_table(page_table *tab, uint addr, uint attrs) {
    tab->present = attrs & 0x1;
    tab->read_write = (attrs >> 1) & 0x1;
    tab->user_supervisor = (attrs >> 2) & 0x1;
    tab->page_level_write_through = (attrs >> 3) & 0x1;
    tab->page_level_cache_disabled = (attrs >> 4) & 0x1;
    tab->accessed = (attrs >> 5) & 0x1;
    tab->dirty_bit = (attrs >> 6) & 0x1;
    tab->page_table_attr_indx = (attrs >> 7) & 0x1;
    tab->global = (attrs >> 8) & 0x1;
    tab->available_11_9 = (attrs >> 9) & 0x3;
    tab->base_address = addr >> 12;
}
```

Ponemos todas las entradas de la page table a la que apunta, en no presente y con atributos en 0. Luego casteamos la entrada del directorio (que inicializamos o ya estaba inicializada) a un struct de tabla de pagina y usamos la posición correspondiente para obtener la entrada en la tabla, una vez que tenemos esto `mmu_inicializar_page_table` pone los atributos correspondientes y la dirección física correspondiente en la entrada de la tabla.

d)

Para habilitar paginación llamamos a la función inicial dirección de kernel que básicamente usando las funciones anteriores mapea el kernel con identity mapping de 0x0 a 0x27000.

```
uint mmu_inicializar_dir_kernel() {
    mmu_inicializar_page_directory((page_directory *)0x27000, 0x28000, 0x3);
    int limpiar = 0;
    for (limpiar = 0x27000 + 0x4; limpiar < 0x28000; limpiar += 0x4)
        mmu_inicializar_page_directory((page_directory *)limpiar, 0x0, 0x0);

    int p_tabla = 0;
    for (p_tabla = 0x0; p_tabla < 0x3FFFFFF; p_tabla += 0x1000)
        mmu_mapear_pagina(p_tabla, 0x27000, p_tabla, 0x3);

    /* mmu_unmapear_pagina(0x3FF000, 0x27000); */

    return 0x27000;
}
```

Esta devuelve el cr3 del kernel (0x270000) el cual lo movemos al cr3 del procesador en `kernel.asm` y activamos paginación con cr0.

```
mov cr3, eax
mov eax, cr0
or  eax, 0x80000000
mov cr0, eax
```

e)

Usamos una función que se comporta parecido a mapear pagina, es decir usamos el cr3 para obtener el directorio y con la parte correspondiente de la dirección virtual obtenemos la entrada a una pagina del directorio y de vuelta usando la dirección virtual obtenemos la entrada en la tabla de paginas correspondiente en la cual seteamos el valor de presente en 0. Esto lo testeamos en la función anterior, donde desmapeamos la última pagina 0x3FF000.

```
uint mmu_unmapear_pagina(uint virtual, uint cr3) {
    cr3 &= 0xFFFFF000;
    page_directory *pd = (page_directory *) cr3;
    uint directorio = (virtual >> 22) & 0x3FF;      /* Posicion en el directorio */
    uint tabla = (virtual >> 12) & 0x3FF;          /* Posicion en la tabla */
    page_table *pt = (page_table *) (pd[directorio].base_adress << 12);
    pt[talbla].present = 0;
    return 0;
}
```

4. Ejercicio 4.

a) La idea es que los perros de cada jugador tengan una memoria compartida entre si. Para eso lo que hicimos fue, pedir al inicio esa memoria para cada jugador, así la tenemos reservada a la hora de crear cada uno de los perros.

```
uint mmu_inicializar() {
    memoria_A = mmu_proxima_pagina_fisica_libre();
    memoria_B = mmu_proxima_pagina_fisica_libre();
}
```

b) Este proceso consta de dos partes, por un lado Poner el perro donde corresponde, como dice el enunciado, y luego asegurarse de que inicie correctamente. Y por otro lado, dejarle definido a dicho perro su esquema de memoria. Inicialmente, lo primero e importante que había que hacer es copiar el código del perro a ejecutarse en memoria, para eso elegimos la dirección 0x7FFF000 que es la última página libre para mapear en el kernel a la parte física en el mapa.

```
uint pagina_a_mapear = 0x7FFF000;
mmu_mapear_pagina(pagina_a_mapear, rcr3(),
    mmu_xy2fisica(perro->jugador->x_cucha, perro->jugador->y_cucha), 0x3);
```

Luego tomamos el código del perro correspondiente:

```
uint codigo_tarea;
int codigo = index_jugador * 10 + index_tipo;
switch (codigo) {
    case (JUGADOR_A*10+ TIPO_1):
        codigo_tarea = 0x10000;
        break;
    case (JUGADOR_A*10+ TIPO_2):
        codigo_tarea = 0x11000;
        break;
    case (JUGADOR_B*10+ TIPO_1):
        codigo_tarea = 0x12000;
        break;
    case (JUGADOR_B*10+ TIPO_2):
        codigo_tarea = 0x13000;
        break;
}
```

copiamos el código del a tarea que tomamos.

```
mmu_copiar_pagina(codigo_tarea, pagina_a_mapear);
```

La memoria ahí copiada puesta va a ser el código del perro desde el inicio de la página hasta que se termina el código. Al final de esa página se va a usar como base para la pila de esa misma tarea.

Ahora, como esto es en 32 bits, la convención C dice que los parámetros se pasaban por pila. Los perros pedían por parámetro el x e y, entonces tomamos la posición de memoria que nosotros mapeamos para copiar el código

```
uint *parametros = (uint *) 0x7FFFFFF8;
```

El valor 0x7FFFFFF8, equivale a una posición de la pila del perro, pero se restó del valor base de la pila por que se envía la dirección de retorno y las dirección de x e y que vamos a escribirle a la tarea perro para que las tome como parámetro.


```
parametros[0] = perro->jugador->x_cucha;  
parametros[1] = perro->jugador->y_cucha;
```

Luego desmapeamos del kernel la pagina donde esta el codigo del perro y su pila

```
mmu_unmapear_pagina(pagina_a_mapear, rcr3());
```

A partir de aca nos encargamos de definir todo el esquema de memoria inicial del perro. Para eso hay que inicializar una page directory. Pedimos una pagina libre para usarla como page directory y la limpiamos.

```
uint pd_perro = mmu_proxima_pagina_fisica_libre();  
mmu_inicializar_pagina((uint *)pd_perro);
```

Hacemos identity mapping del kernel y el area libre:

```
for (p_tabla = 0x0; p_tabla <= 0x3FFFFFF; p_tabla += 0x1000)  
    mmu_mapear_pagina(p_tabla, pd_perro, p_tabla, 0x1);
```

Mapeamos la direccion virtual a la direccion fisica, con permiso de lectura solo para q se pueda saber q esta ahi, pero no escribirse a si mismo.

```
mmu_mapear_pagina(mmu_xy2virtual(perro->jugador->x_cucha, perro->jugador->y_cucha),  
    pd_perro, mmu_xy2fisica(perro->jugador->x_cucha, perro->jugador->y_cucha), 0x5);
```

Mapeamos la memoria compartida del perro con los otros perros, dependiendo del jugador, usando la variable global que habiamos iniciado anteriormente:

```
// mapear dir compartida a 0x400000  
if (index_jugador == JUGADOR_A) {  
    mmu_mapear_pagina(0x400000, pd_perro, memoria_A, 0x7);  
} else {  
    mmu_mapear_pagina(0x400000, pd_perro, memoria_B, 0x7);  
}
```

Y por ultimo le mapeamos al perro la direccion 0x401000 virtual a donde va a estar fisicamente su codigo, asi, luego cuando se mueva por el mapa, se va mover su codigo y su pila, y se va a remapear la nueva posicion a 0x401000, asi el perro siempre busca a la misma direccion su siguiente instruccion y su pila.

```
mmu_mapear_pagina(0x401000, pd_perro,  
    mmu_xy2fisica(perro->jugador->x_cucha, perro->jugador->y_cucha), 0x7);
```

5. Ejercicio 5.

Completamos el código para las interrupciones ISR32, ISR33 y isr70 que son las interrupciones de reloj, de teclado y la interrupción 46 respectivamente .

```
_isr32:
    pushad
    call fin_intr_pic1
    cmp dword [debug_mode],0
    je .continuar
    cmp dword [debug_view], 1
    je .fin
.continuar:
    call sched_atender_tick
    str cx
    shl ax, 3
    cmp ax, cx
    je .fin
    mov [sched_tarea_selector], ax
    jmp far [sched_tarea_offset]
.fin:

    popad
    iret
```

Este código guarda el estado de los registros, llama a la función `fin_intr_pic1` , luego llama a `sched_atender_tick` , que es una función que va a atender esta interrupción. Luego salta a la tarea Idle.

Análogamente la interrupción para teclado:

```
_isr33:
    pushad
    call fin_intr_pic1
    in al, 0x60
    push eax
    call atender_teclado
    pop eax
    popad
    iret
```

Donde `atender_teclado` va a ser la función que atienda la interrupción de teclado. Esta función recibe por parámetro el código de carácter que está siendo presionado, el cual pusheamos a la pila.

La interrupción 70 llama a la función `game_atender_teclado` enviándoles los parámetros por pila.

```
_isr70:
    push ecx
    push edx
    push ebx
    push ebp
    push esi
    push edi

    push ecx
    push eax
    call game_atender_pedido
    jmp 0x70:0
    add esp, 8

    pop edi
    pop esi
    pop ebp
    pop ebx
    pop edx
    pop ecx
    iret
```

Luego completamos las funciones `sched_atender_tick` y `atender_teclado`.

```
ushort sched_atender_tick() {
    game_atender_tick(scheduler.tasks[scheduler.current].perro);
    scheduler.current = sched_proxima_a_ejecutar();
    game_perro_actual = scheduler.tasks[scheduler.current].perro;
    return scheduler.tasks[scheduler.current].gdt_index;
}
```

Dentro de la funcion `game_atender_tick` , se llama a la funcion `mostrar_reloj` que esta defininida en `screen.c`

```
void mostrar_reloj() {
    if(contador_reloj==5){
        contador_reloj = 0;
    }
    char c = reloj[contador_reloj];
    contador_reloj++;
    screen_pintar_rect(c, C_FG_WHITE, 0, 79, 1, 1);
}
```

Esta funcion usa la variable global `contador_reloj` la cual se inicializa en 0, e itera por los caracteres definidos en el array `reloj`, luego llama a la funcion `screen_pintar_rect` la cual pinta en pantalla el caracter seleccionado, de color blanco en la posicion (0,79), en un cuadrado de 1x1.

La funcion `atender_teclado` tambien definida en `screen.h`, es basicamente un switch el cual dependiendo de que caracter llega por parametro, llama a la funcion `pintar_atender_teclado` con el codigo ascii correspondiente a esa tecla. En caso de presionarse la Y, se entra en llama a la rutina de atencion de debug.

```
void atender_teclado(unsigned char tecla){
    switch (tecla) {
        case KB_q: game_jugador_lanzar_perro(&jugadorA, TIPO_1, jugadorA.x_cucha, jugadorA.y_cucha); break;
        case KB_e: game_jugador_lanzar_perro(&jugadorA, TIPO_2, jugadorA.x_cucha, jugadorA.y_cucha); break;
        case KB_u: game_jugador_lanzar_perro(&jugadorB, TIPO_1, jugadorB.x_cucha, jugadorB.y_cucha); break;
        case KB_o: game_jugador_lanzar_perro(&jugadorB, TIPO_2, jugadorB.x_cucha, jugadorB.y_cucha); break;
        case KB_w: game_jugador_moverse(&jugadorA,0, -1); break;
        case KB_a: game_jugador_moverse(&jugadorA, -1, 0); break;
        case KB_s: game_jugador_moverse(&jugadorA, 0, 1); break;
        case KB_d: game_jugador_moverse(&jugadorA, 1, 0); break;
        case KB_i: game_jugador_moverse(&jugadorB, 0, -1); break;
        case KB_j: game_jugador_moverse(&jugadorB, -1, 0); break;
        case KB_k: game_jugador_moverse(&jugadorB, 0, 1); break;
        case KB_l: game_jugador_moverse(&jugadorB, 1, 0); break;
        case KB_z: game_jugador_dar_orden(&jugadorA, 1); break;
        case KB_x: game_jugador_dar_orden(&jugadorA, 2); break;
        case KB_c: game_jugador_dar_orden(&jugadorA, 3); break;
        case KB_b: game_jugador_dar_orden(&jugadorB, 1); break;
        case KB_n: game_jugador_dar_orden(&jugadorB, 2); break;
        case KB_m: game_jugador_dar_orden(&jugadorB, 3); break;
        case KB_y: atender_debug(); break;

    }
}
```

6. Ejercicio 6.

a), b), d), e)

Para este punto definimos una tss idle, una tss inicial y las tss para todos los jugadores por medio de

```
tss tss_inicial;
tss tss_idle;

tss tss_jugadorA[MAX_CANT_PERROS_VIVOS];
tss tss_jugadorB[MAX_CANT_PERROS_VIVOS];
```

Donde la máxima cantidad es 8. Al tener esto podemos inicializar las entradas de descriptors de la gdt, desde la 13 a la 30, con 13 la tarea inicial, 14 la tarea idle y el resto tareas para los jugadores primero las del jugador A y luego las del jugador B. Hacemos esto con la función `tss_inicializar` que vamos a llamar desde `kernel.asm`.

```
void tss_inicializar() {
    gdt[GDT_TSS_TAREA_INICIAL] = (gdt_entry) {
        (unsigned short)    TSS_KERNEL_LIMIT & 0xffff,
        (unsigned short)    (unsigned int) (& tss_inicial) & 0xffff,
        (unsigned char)     ((unsigned int) (& tss_inicial) >> 16) & 0xff,
        (unsigned char)     0x9,
        (unsigned char)     0x0,
        (unsigned char)     0x0, /* dpl */
        (unsigned char)     0x1,
        (unsigned char)     (TSS_KERNEL_LIMIT >> 16) & 0xf,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     ((unsigned int) (& tss_inicial) >> 24) & 0xff,
    };

    gdt[GDT_TSS_IDLE] = (gdt_entry) {
        (unsigned short)    TSS_KERNEL_LIMIT & 0xffff,
        (unsigned short)    (unsigned int) (& tss_idle) & 0xffff,
        (unsigned char)     ((unsigned int) (& tss_idle) >> 16) & 0xff,
        (unsigned char)     0x9,
        (unsigned char)     0x0,
        (unsigned char)     0x0, /* dpl */
        (unsigned char)     0x1,
        (unsigned char)     (TSS_KERNEL_LIMIT >> 16) & 0xf,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     ((unsigned int) (& tss_idle) >> 24) & 0xff,
    };
}
```

Ponemos los dpls de todas las tareas en 0. Esto nos va a permitir que las tareas no se llamen unas a otras, si su CPL no es 0, lo cual va a pasar efectivamente para las tareas de los jugadores. Es decir el cambio de contexto va a tener que pasar durante otro nivel de privilegio que va a ser el del reloj. Como base ponemos la dirección de la tss correspondiente en memoria y como límite 103 bytes (cada tss ocupa 104). Luego llenamos las entradas para todas las tareas y llamamos a una función que inicializa la tss de la tarea idle (siguiente pagina).

```

int i = 0;

/* GDT e indice jugador A */
for (i = 0; i < MAX_CANT_PERROS_VIVOS; i++) {
    indices_A[i]=FALSE;
    gdt[entrada_libre] = (gdt_entry) {
        (unsigned short)    TSS_KERNEL_LIMIT & 0xffff,
        (unsigned short)    (unsigned int) (&tss_jugadorA[i] & 0xffff,
        (unsigned char)     ((unsigned int) (&tss_jugadorA[i]) >> 16) & 0xff,
        (unsigned char)     0x9,
        (unsigned char)     0x0,
        (unsigned char)     0x0,                                /* dpl          */
        (unsigned char)     0x1,
        (unsigned char)     (TSS_KERNEL_LIMIT >> 16) & 0xf,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     ((unsigned int) (&tss_jugadorA[i]) >> 24) & 0xff,
    };
    entrada_libre++;
}

/* GDT e indice jugador B */
for (i = 0; i < MAX_CANT_PERROS_VIVOS; i++) {
    indices_B[i]=FALSE;
    gdt[entrada_libre] = (gdt_entry) {
        (unsigned short)    TSS_KERNEL_LIMIT & 0xffff,
        (unsigned short)    (unsigned int) (&tss_jugadorB[i] & 0xffff,
        (unsigned char)     ((unsigned int) (&tss_jugadorB[i]) >> 16) & 0xff,
        (unsigned char)     0x9,
        (unsigned char)     0x0,
        (unsigned char)     0x0,                                /* dpl          */
        (unsigned char)     0x1,
        (unsigned char)     (TSS_KERNEL_LIMIT >> 16) & 0xf,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     0x0,
        (unsigned char)     ((unsigned int) (&tss_jugadorB[i]) >> 24) & 0xff,
    };
    entrada_libre++;
}

//Inicializando la tss de Idle.
completar_tss_idle();
}

```

```
void completar_tss_idle() {
    tss_idle.unused0      = 0;
    tss_idle.esp0        = 0x27000;
    tss_idle.ss0         = 0x50;
    tss_idle.unused1      = 0;
    tss_idle.esp1        = 0;
    tss_idle.ss1         = 0;
    tss_idle.unused2      = 0;
    tss_idle.esp2        = 0;
    tss_idle.ss2         = 0;
    tss_idle.unused3      = 0;
    tss_idle.cr3         = k_cr3;
    tss_idle.eip          = 0x16000;
    tss_idle.eflags       = 0x202;
    tss_idle.esp          = 0x27000;
    tss_idle.ebp          = 0x27000;
    tss_idle.es           = 0x50;
    tss_idle.unused4      = 0;
    tss_idle.cs           = 0x40;
    tss_idle.unused5      = 0;
    tss_idle.ss           = 0x50;
    tss_idle.unused6      = 0;
    tss_idle.ds           = 0x50;
    tss_idle.unused7      = 0;
    tss_idle.fs           = 0x50;
    tss_idle.unused8      = 0;
    tss_idle.gs           = 0x50;
    tss_idle.unused9      = 0;
    tss_idle.unused10     = 0;
    tss_idle.iomap        = 0xFFFF;
}
```

Hacemos apuntar su cr3 al del kernel como lo pedia el enunciado y el eip en 0x16000 que es donde esta definida la tarea idle (idle.asm). Además hacemos apuntar todos sus selectores de segmento, menos cs, a la decima entrada de la gdt la cual es la de datos de nivel 0 y cs a codigo de nivel 0.

c)

Para este punto escribimos la siguiente función

```
void llenar_descriptor_tss_perro(int indice, perro_t *perro, int index_jugador, int index_tipo) {
    if (index_jugador == JUGADOR_A) {
        tss_jugadorA[indice].unused0 = 0;
        tss_jugadorA[indice].esp0 = mmu_proxima_pagina_fisica_libre();
        tss_jugadorA[indice].ss0 = 0x5B;
        tss_jugadorA[indice].unused1 = 0;
        tss_jugadorA[indice].esp1 = 0;
        tss_jugadorA[indice].ss1 = 0;
        tss_jugadorA[indice].unused2 = 0;
        tss_jugadorA[indice].esp2 = 0;
        tss_jugadorA[indice].ss2 = 0;
        tss_jugadorA[indice].unused3 = 0;
        tss_jugadorA[indice].cr3 = mmu_inicializar_memoria_perro(perro, index_jugador, index_tipo);
        tss_jugadorA[indice].eip = 0x401000;
        tss_jugadorA[indice].eflags = 0x202;
        tss_jugadorA[indice].esp = 0x401000 + 0x1000 - 12;
        tss_jugadorA[indice].ebp = 0x401000 + 0x1000 - 12;
        tss_jugadorA[indice].es = 0x5B;
        tss_jugadorA[indice].unused4 = 0;
        tss_jugadorA[indice].cs = 0x4B;
        tss_jugadorA[indice].unused5 = 0;
        tss_jugadorA[indice].ss = 0x5B;
        tss_jugadorA[indice].unused6 = 0;
        tss_jugadorA[indice].ds = 0x5B;
        tss_jugadorA[indice].unused7 = 0;
        tss_jugadorA[indice].fs = 0x5B;
        tss_jugadorA[indice].unused8 = 0;
        tss_jugadorA[indice].gs = 0x5B;
        tss_jugadorA[indice].unused9 = 0;
        tss_jugadorA[indice].unused10 = 0;
        tss_jugadorA[indice].iomap = 0xFFFF;
    }
}
```

Esta función toma como parametros un indice (indice en tss_jugadorX) un puntero a un perro, el index del jugador, si es A o B y el tipo de perro. Además, a diferencia de la inicial y la idle los selectores de segmento de las tareas perros apuntan a código y nivel 3. Es decir su CPL va a ser 3 y no 0, y como las entradas de la GDT de sus descriptors tienen 0 evitan que pueda ser posible que un perro salte a otro perro sin pasar por el reloj. Eip esta siempre fijo, ya que su esquema de paginación va a ser el que defina donde esta la tarea al igual que su pila, donde el -12 es porque pusha su dirección de retorno y los eflags. Cada perro tiene su cr3 que es dado por mmu_inicializar_memoria_perro (Ver punto 4).

f)

Para este punto pusimos todas las funciones que definimos en los puntos anteriores como extern en kernel.asm, llamamos a tss_inicializar y luego agregamos las siguientes instrucciones:

```
mov bx, 0x68
ltr bx
;=====
jmp 0x70:0
```

Es decir, cargamos al task register, por medio de un registro en este caso bx (o una dirección en memoria) el selector de la tarea inicial que es 0x68 equivalente a la entrada 13 de la gdt. Luego simplemente hacemos un jmp (far) con el selector igual a la entrada 14, es decir 0x70 en hexadecimal, de la tarea idle.

g)

Para la syscall 46 escribimos lo siguiente en isr.asm

```
global _isr46:
_isr46:
    pushad
    call fin_intr_pic1
    push ecx
    push eax
    call game_atender_pedido
    popad
    iret
```

Pusheamos por convención C 32 bits, primero ecx y luego eax. El ecx lo vamos a usar solo para una de las 4 llamadas posibles que va a hacer la función `game_atender_pedido`. En los otros casos no se usa. La función atender pedido es la siguiente y hace lo que se espera, llamar a las funciones correspondientes acorden a la orden dada por el perro, `game_perro_actual` es una variable global que guarda el puntero al perro que mando la orden. La función `olfatear` esta dada por la cátedra. Para devolver la última orden dada por el jugador (`eax == 4`) usamos el struct del perro que apunta a un jugador con la última orden dada

```
uint game_atender_pedido(int eax, int ecx){
    if(eax == 1)
        game_perro_mover(game_perro_actual, ecx);
    if(eax == 2)
        game_perro_cavar(game_perro_actual);
    if(eax == 3)
        game_perro_olfatear(game_perro_actual);
    if(eax == 4)
        return (game_perro_actual->jugador)->index ? jugadorB.orden : jugadorA.orden;
    return 0;
}
```

`Perro_cavar` ejecuta el siguiente código

```
uint game_perro_cavar(perro_t *perro){
    if(game_parado_en_escondite(perro->x, perro->y)
        && game_huesos_en_posicion(perro->x, perro->y)
        && (perro->huesos < 10))
    {
        game_sacar_hueso(perro->x, perro->y, perro);
    }
    return 0;
}
```

Donde `parado_en_escondite`, `huesos_en_posicion` y `sacar_huesos` son las siguientes funciones

```

uint game_parado_en_escondite(uint x, uint y){
    int *escondite;
    escondite = game_dame_escondite(x ,y);
    return escondite != NULL ? 1 : 0;
}

uint game_huesos_en_posicion(uint x, uint y){
    int *escondite;
    escondite = game_dame_escondite(x ,y);
    return escondite != NULL ? escondite[3] : 0;
}

void game_sacar_hueso(uint x, uint y, perro_t * perro){
    int *escondite;
    escondite = game_dame_escondite(x ,y);
    escondite[3]--;
    perro->huesos++;
}

```

Devuelven lo que se espera, `parado_en_escondite` llama a `dame_escondite` que de haber un escondite en la posición devuelve un puntero al escondite, `huesos_en_posicion` usa también la función `dame_escondite` y en caso de ser no null devuelve los huesos en la posición y de lo contrario 0, y `game_sacar_hueso` usa el puntero ese para restarle los huesos al escondite y sumárselos al perro correspondiente. Además nos fijamos que el perro tenga menos de 10 huesos para cavar. La función `dame_escondite` es la siguiente:

```

int* game_dame_escondite(uint x, uint y){
    int i;
    for(i = 0; i < ESCONDITES_CANTIDAD; i++){
        if(escondites[i][1] == x && escondites[i][2] == y)
            return escondites[i];
    }
    return NULL;
}

```

La cual recorre el arreglo de escondites mirando su subarreglo (x, y, huesos) y si la posición es la que se le paso por parametro devuelve la dirección del escondite.

7. Ejercicio 7.

7.1. a)

Para el funcionamiento del scheduler lo que usamos fue una variable global scheduler que contenía una lista de tareas, que consistían en punteros a perro y el índice de la gdt de la tarea. Y el índice de esa lista correspondiente a la tarea que se esté ejecutando en ese momento.

Para la lista de tareas definimos una array de 17 elementos, donde el elemento 0 era el de la tarea IDLE. Los índices del 1 al 8 corresponden a los perros del jugador A y por último los índices del 9 al 16 corresponden al del jugador B.

Iniciamos los valores para el índice de la tarea IDLE, y dejamos nulos los valores de índice de gdt y el puntero a perro, luego se va a usar para saber si hay un perro corriendo en ese índice a partir del valor nulo o no del índice de gdt

```
scheduler.tasks[0].gdt_index = GDT_TSS_IDLE;
scheduler.tasks[0].perro = NULL;
for (i = 1; i < MAX_CANT_TAREAS_VIVAS + 1; i++) {
    scheduler.tasks[i].gdt_index = NULL;
    scheduler.tasks[i].perro = NULL;
}
scheduler.current = 0;
```

7.2. b)

Para la función próxima a ejecutar definimos una función auxiliar que nos daba el próximo perro (índice de la tarea) del jugador siguiente y de no haberlo buscábamos en los perros del jugador actual y en caso de que los dos no tuviesen perros corriendo defaulteábamos a la tarea idle. El código es el siguiente.

```
uint sched_proxima_a_ejecutar() {
    uint prox_jugador = (scheduler.current <= 8);
    uint prox_task = sched_proximo_perro_jugador(prox_jugador);

    if (!prox_task) {
        prox_jugador = !prox_jugador;
        prox_task = sched_proximo_perro_jugador(prox_jugador);
        if (!prox_task)
            return 0;
    }
    ultimo_index[prox_jugador] = prox_task;
    return prox_task;
}
```

y la función sched_proximo_perro_jugador es

```
uint sched_proximo_perro_jugador(uint jugador) {
    uint index = ultimo_index[jugador];
    do {
        index++;
        if (index > (8 << jugador))
            index -= 8;
        if (scheduler.tasks[index].gdt_index != NULL)
            return index;
    } while (index != ultimo_index[jugador]);
    return 0;
}
```

Donde básicamente loopeamos desde la ultima tarea ejecutada (sin considerarla) en adelante, si no encontramos tareas en ese intervalo, restamos 8 y empezamos desde el principio hasta la ultima tarea ejecutada inclusive, en caso de no encontrar una tarea libre devolvemos 0. notar que $8 \ll \text{jugador}$ da el valor correcto en el que restamos 8, si el jugador es el A, es decir el 0, restamos 8 cuando index sea igual a 9, y si es el jugadorB es decir el 1, el shift hace que 8 sea 16 y restamos cuando index es 17.

7.3. c) y e)

Se setea la tarea que se esta ejecutando como la nueva tarea a ser ejecutada.

```
scheduler.current = sched_proxima_a_ejecutar();
game_perro_actual = scheduler.tasks[scheduler.current].perro;
```

En la interrupcion de reloj se agrego el siguiente codigo, tomando la proxima tarea a ejecutar y haciendo la conmutacion de tareas

```
call sched_atender_tick
;xor ecx, ecx
str cx
shl ax, 3
cmp ax, cx
je .fin
mov [sched_tarea_selector], ax
jmp far [sched_tarea_offset]
```

7.4. d)

Se cambio un poco el codigo de la interrupcion para poder devolver EAX como resultado, Asi como tambien el jmp a la tarea idle, usando 0x70 como selector, dado que $0x70 = 1110000b$, si tomamos la parte del indice nos queda 1110 que equivale a 14 en decimal, el indice de la gdt de la tarea IDLE

```
push ecx
push edx
push ebx
push ebp
push esi
push edi

push ecx
push eax
call game_atender_pedido
jmp 0x70:0
add esp, 8

pop edi
pop esi
pop ebp
pop ebx
pop edx
pop ecx
iret
```

Para mover el perro habia que chequear que la nueva posicion fuese valida

```

int x, y;
game_dir2xy(dir, &x, &y);
int nuevo_x = perro->x + x;
int nuevo_y = perro->y + y;

if(!game_es_posicion_valida(nuevo_x, nuevo_y))
    return 0;

if(game_perro_en_posicion_j(perro->jugador, nuevo_x, nuevo_y) != NULL)
    return 0;

```

Se tiene que hacer el proceso de mover el perro en si, mapeamos la nueva direccion y copiamos entero al codigo del perro y su pila , luego cambiamos en paginacion la traduccion de la direccion 0x401000 a la nueva posicion escrita. y por ultimo mapear la posicion del perro solo como lectura.

```

uint dir_fisica = mmu_xy2fisica(nuevo_x, nuevo_y);
uint dir_virtual = mmu_xy2virtual(nuevo_x, nuevo_y);

mmu_mapear_pagina(0x7FFF000, rcr3(), dir_fisica, 0x3);
mmu_copiar_pagina(0x401000, 0x7FFF000);

mmu_mapear_pagina(0x401000, rcr3(), dir_fisica, 0x7);
mmu_mapear_pagina(dir_virtual, rcr3(), dir_fisica, 0x5);

mmu_unmapear_pagina(0x7FFF000, rcr3());

```

Refrescamos en la pantalla la nueva posicion del perro, y nos fijamos si esta en la cucha

```

screen_borrar_perro(perro);
perro->x = nuevo_x;
perro->y = nuevo_y;
screen_pintar_perro(perro);
game_perro_ver_si_en_cucha(perro);
return 1;

```

En caso de estarlo, anota los puntos y luego termina la tarea del perro

```

void game_perro_ver_si_en_cucha(perro_t *perro)
{
    if (perro->x != perro->jugador->x_cucha || perro->y != perro->jugador->y_cucha){
        return;
    }

    if (perro->huesos == 0){
        return;
    }

    while(perro->huesos > 0){
        game_jugador_annotar_punto(perro->jugador);
        perro->huesos--;
    }

    game_perro_termino(perro);
}

```

```

void game_jugador_anotar_punto(jugador_t *j) {
    ultimo_cambio = MAX_SIN_CAMBIOS;

    j->puntos++;

    screen_pintar_puntajes();

    if (j->puntos == 999)
        screen_stop_game_show_winner(j);
}

```

```

void game_perro_termino(perro_t *perro){
    sched_remove_tarea(perro->id);
}

```

Para sacar al perro que ya termino, buscamos el indice dentro de la lista de tareas a partir del `gdt_index` recibido, borra al perro de la pantalla, libera al perro y saca al perro del scheduler

```

void sched_remove_tarea(unsigned short gdt_index) {

    uint i = 1;
    while (scheduler.tasks[i].gdt_index != gdt_index)
        i++;
    perro_t *p = scheduler.tasks[i].perro;
    screen_borrar_perro(p);
    p->libre = TRUE;
    screen_pintar_reloj_perro(p);
    scheduler.tasks[i].gdt_index = NULL;
    if (i == scheduler.current)
        scheduler.current = NULL;
}

```

g)

Para este punto, se agrego las variables *debugMode* y *debugView*. La primera indica si se entro en modo debug, y la segunda indica si en este momento se esta mostrando informacion en la pantalla.

Luego se cambiaron tanto la rutina de atencion de interrupciones de teclado, para agregar la tecla Y, la cual setea estas variables, como el macro para la atencion de excepciones. Este se modifico para que en caso de encontrarse en modo debug, se cree una copia de la pantalla, y luego se proceda a mostrar la informacion de la excepcion requerida por el enunciado.

Se utilizo un arreglo para mantener una copia de la pantalla anterior.

```

short pantalla[80 * 50];
.....
.....
void game_guardar_pantalla() {
    short *src = (short *)0xB8000;
    int i;
    for(i = 0; i < 80 * 50; i++){
        pantalla[i] = src[i];
    }
}

```

La rutina de atención de interrupciones del reloj, se modifico para que en caso de encontrarse en debug, no se ejecute el salto de tarea sino que se espera hasta que se desactive el mismo.

```
    cmp dword [debug_mode],0
    je .continuar
    cmp dword [debug_view], 1
je .fin
.continuar:
    ....
    rutina de atencion reloj
    ....
.fin:
    popad
    iret
```

Para escribir los estados de los flags, como de los registros, los mismos se pushean a la pila y se pasan por parametro para luego imprimirlos por pantalla.

```
mov eax, [esp]
pushf
push eax
call game_imprimir_info_debug
```