DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Zombi Defense

Organización del Computador II

Primer Cuatrimestre de 2017

Integrante	LU	Correo electrónico
Franco Martinez Quispe	025/14	francogm01@gmail.com
Mirko Yves Torrico	28/10	mirko.torrico@gmail.com
Carlos Daniel Núñez Morales	732/08	cdani.nm@gmail.com

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

Índice

1. Ejercicio 1

En este ejercicio creamos dos descriptores de datos y dos de código, e inicializamos la GDT con ellos, también le agregamos un descriptor para el segmento de video. Luego pasamos a modo protegido, seteamos la pila en la dirección 0X27000 y finalmente probamos que el descriptor de video sea correcto usandolo para escribir en pantalla.

1.1. Ej 1.a)

En este item lo que hicimos fue inicializar la GDT. Por restricción del trabajo práctico las 7 primeras posiciones se consideran usadas. En ese sentido cargaremos los descriptores a partir del índice 8 de la siguiente manera: en el indice 8 y 9, los descriptores de codigo y datos para kernel (respectivamente); en el 10 y 11 los descriptores de código y datos para usuario (respectivamente); el índice 12 de la GDT será para el descriptor de video. Para realizar esto usamos el arreglo de descriptores (GDT) provisto por la cátedra cuyo código se encuentra en $gdt.h$ y $gdt.c$, seteamos los valores adecuadamente para las posiciones del 8 al 12 (para indexar el arreglo usamos constantes definidas en $defines.c$).

```
\\a partir de la entrada 8 de la gdt empezamos a completar
[kernel_code] = (gdt_entry) {
        (unsigned short)    0x6eff,         /* limit[0:15]  */
        (unsigned short)    0x0000,         /* base[0:15]   */
        (unsigned char)     0x00,           /* base[23:16]  */
        (unsigned char)     0x0A,           /* type         */
        (unsigned char)     0x01,           /* s            */
        (unsigned char)     0x00,           /* dpl          */
        (unsigned char)     0x01,           /* p            */
        (unsigned char)     0x02,           /* limit[16:19] */
        (unsigned char)     0x00,           /* avl          */
        (unsigned char)     0x00,           /* l            */
        (unsigned char)     0x01,           /* db           */
        (unsigned char)     0x01,           /* g            */
        (unsigned char)     0x00,           /* base[31:24]  */

    },
```

Le siguen el descriptor de segmento para datos de kernel (nivel 0), Luego los dos descriptores de código y datos (nivel 3) para usuario, dirección base y limite son exactamente igual.

Aclaración: Para el descriptor de segmento de nivel 0 ponemos el campo *tipo* en $0x0A$ (1010) y el bit *S* en 1 ya que no es de sistema (es de código o datos). El bit de presente lo ponemos en 1 porque el segmento esta presente. Como trabajamos con 32 bits ponemos *d/b* en 1 y el bit *l* en 0.
El segmento de datos es igual que el segmento de código, salvo por el campo *tipo*, que en este caso es 0x2 (0010) porque es un segmento de datos que se puede leer y escribir.

1.2. Ej 1.b)

Pasando a modo protegido y seteando la pila: Para eso, una vez cargada la gdt, deshabilitamos las interrupciones, cargamos el registro $GDTR$, seteamos el bit PE del registro $CR0$, se cragan los registros de segmento

```
; Habilitar A20
  call habilitar_A20
  ; Cargar la GDT
  lgdt [GDT_DESC]
  ; Setear el bit PE del registro CR0
  mov eax, CR0
```

```
    or eax, 1
    mov CR0, eax
    ; Saltar a modo protegido
    jmp 0x40:modo_protegido

BITS 32
modo_protegido:
    xor eax, eax
    ; Establecer selectores de segmentos 00001001000
    mov ax, 0x48 ;indice 9 de GDT
    mov ss, ax
   ; jmp $
    mov ds, ax
    mov gs, ax
    mov fs, ax
    mov ax, 0x060 ;indice 12 de GDT
    mov es, ax
;0000 1100 000
    ; Establecer la base de la pila
    mov esp, 0x27000
```

Aclaración: Lo que hacemos aquí es mover $cr0$ a eax para poder setear el bit de modo protegido (bit PE) en 1, y luego moverlo nuevamente a $cr0$. Luego cambiamos el selector de segmento porque el procesador había preparado las instrucciones en modo real, la manera para pasar a modo protegido es realizando un $jump$, para que las ejecute. Saltamos al segmento de código a la etiqueta de $modoprotegido$. Movemos los índices adecuados a los selectores adecuados(cs, ds, es, gs, ss, fs). También inicializamos la pila en la dirección 0x27000 como lo pide el enunciado.

1.3. Ej1 c y d)

Se pide declarar un segmento adicional que describa el área de pantalla en memoria y una rutina para escribir en pantalla

```
[pantalla] = (gdt_entry) {
        (unsigned short)    0x1f3f,         /* limit[0:15]  */
        (unsigned short)    0x8000,         /* base[0:15]   */
        (unsigned char)     0x0B,           /* base[23:16]  */
        (unsigned char)     0x03,           /* type         */
        (unsigned char)     0x01,           /* s            */
        (unsigned char)     0x03,           /* dpl          */
        (unsigned char)     0x01,           /* p            */
        (unsigned char)     0x00,           /* limit[16:19] */
        (unsigned char)     0x00,           /* avl          */
        (unsigned char)     0x00,           /* l            */
        (unsigned char)     0x01,           /* db           */
        (unsigned char)     0x00,           /* g            */
        (unsigned char)     0x00,           /* base[31:24]  */
```

Finalmente el segmento de video. Aquí lo que haremos es poner la base en 0xb8000 dado que es en esa dirección donde empieza la memoria de video. Luego se calcula el límite: son 50 filas y 80 columnas de dos bytes cada posición por eso el tamaño es 80*50*2 = 8000 = 0x1f40, el límite es uno menos ya que empezamos a contar desde el 0 y sería el último byte efectivo para acceder, por lo tanto queda 0x1f3f. El campo $tipo$ queda en $0x03$ (segmento de datos que se puede leer y escribir) y el bit G lo ponemos en 0 ya que el límite no es un número muy grande. El resto de los campos se mantienen iguales al segmento

de Usuario antes mensionados.
Nos quedaría por lo tanto una GDT de la siguiente manera:

Descripción	Index	Base	Limite	Tipo	S	P	DPL	AVL	L	D/B	G
nulo	00	0x0	0x0	0x0	0	0	0	0	0	0	0
reservados	1..7	0x0	0x0	0x0	0	0	0	0	0	0	0
kernel_code	8	0x0	0x26eFF	0x0A	1	1	0	0	0	1	1
kernel_data	9	0x0	0x26eFF	0x02	1	1	0	0	0	1	1
user_code	10	0x0	0x26eFF	0x0A	1	1	3	0	0	1	1
user_data	11	0x0	0x26FFF	0x02	1	1	3	0	0	1	1
video	12	0xb8000	0x5fff	0x03	1	1	3	0	0	1	0

2. Ejercicio 2

En este ejercicio nos encargamos de crear las entradas necesarias de la IDT y de su correspondiente implementación para poder atender las excepciones que genera el procesador.
Lo primero que hicimos fue completar las entradas de la IDT en el archivo $idt.c$.

```
//codigo de idt.c
#define IDT_ENTRY_SYS(n)
idt[n].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## n) & (unsigned int) 0xFFFF);
idt[n].segsel = (unsigned short) 0x40; //seg kernel codigo en GDT
idt[n].attr = (unsigned short) 0xee00;  //interrup gate
idt[n].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## n) >> 16 & (unsigned int) 0xFFFF);

#define IDT_ENTRY(n)
idt[n].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## n) & (unsigned int) 0xFFFF);
idt[n].segsel = (unsigned short) 0x40;  //seg kernel codigo en GDT
idt[n].attr = (unsigned short) 0x8e00; //interrup gate
idt[n].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xFFFF

//se usa las definiciones de arriba para poder completar el metodo idt_inicializar()
//asi que creamos las entradas para los primeros 19 descriptores de interrupción luego el 32 y 33
void idt_inicializar() {
        IDT_ENTRY(0);
        ...
        //(del 0 al 19)
        ...
        IDT_ENTRY(19);
        IDT_ENTRY(32);
        IDT_ENTRY(33);
        IDT_ENTRY_SYS(120);
```

Para poder atender excepciones (e interrupciones más adelante) se completó el archivo $isr.h$ agregando la siguiente declaración de función ”void_isrX()”donde X es el número de excepción declarada para su siguiente implementación.
La implementación de las mismas se llevó acabo en el archivo $isr.asm$. Y se probó imprimiendo en pantalla el número de excepción.

Para poder atender las interrupciones del 0 al 19. La catedra nos facilito el siguiente macro, con el numero de interrupcion en eax:

```
%macro ISR 1
global _isr%1
```

```
_isr%1:

    push eax
    resolver_excepcion
jmp $
%endmacro
```

Como tenemos el numero de excepcion/interrupcion en eax, llamamos a la **resolver_excepcion**, la cual nos ayuda a identificar la excepcion que se produjo, imprimiendo en la parte superior de la pantalla cual excepcion se produjo.
La aridad y funcion es la siguiente

```
void resolver_excepcion(int i)
{
switch(i){
            case 0:
                print("Divide Error",20,0,C_FG_WHITE);
                break;
            case 1:
                print("Reserved", 20,0,C_FG_WHITE);
                break;
            .. /* idem para todas las otras 17 excepciones de intel  */
}
```

3. Ejercicio 3

3.1. Ej3 a)

Para este ítem se escribió una rutina para limpiar y setear la pantalla de video como figura en la imagen 9 del enunciado.
La rutina se escribió en *screen.c* y tiene 2 procedimientos: **print_black_screen** que limpia la pantalla (la pone de color negro) y luego **print_game_screen** le da el color a cada posición de la pantalla para que quede como muestra la figura.

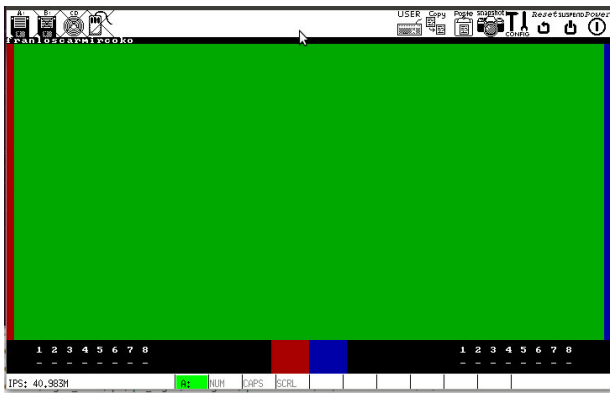

Figura 1: Imagen de la pantalla

3.2. Ej3 b)

En este ejercicio nos piden inicializar el directorio y tabla de páginas para el kernel antes de activar paginación.
En el archivo *mmu.c* tenemos la función *mmu_inicializar_dir_kernel*() definida de la siguiente manera:

Algorithm 1 inicializar_dir_kernel()

```
1: PageDirectory *PD ← castear_como_puntero(0x27000)
2: PageTable *PT ← castear_como_puntero(0x28000)
3: //inicializamos la primer entrada del PageDirectory
4: PD[0].dir_base ← (PT >> 12)
5: PD[0].rw / p ← 1 / 1
6: //todas los demás atributos de PD[0] se setean con cero
7: //y todas las demás entradas del PD se llenan con ceros (Bit P como no presente)
8: for i←1, hasta 1023 do
9:    PD[i] ← 0x00000000
10: end for
11: //luego seteamos TODAS las entradas de la PageTable (con identity_mapping)
12: unsigned_int dir_fisica ← 0
13: for i←0, hasta 1023 do
14:    PT[i].dir_base ← (dir_fisica >> 12)
15:    PT[i].rw / p ← 1 / 1
16:    //todos los demás atributos de PT[i] se setean con cero
17:    dir_fisica ← dir_fisica + 1
18: end for
19: tlbflush()
```

Hasta acá queda inicializada la paginación para el kernel con identity mapping desde la dirección 0x00000000 hasta 0x003fffff.

3.3. Ej3 c)

Luego se activa paginación desde $kernel.asm$:

```
; Inicializar el manejador de memoria
    call mmu_inicializar_dir_kernel
    ; Cargar directorio de paginas
    mov eax, 0x27000
    mov CR3, eax
    ; Habilitar paginacion
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax
```

4. Ejercicio 4

4.1. Ej4 a)

Para poder administrar debidamente la memoria en el sistema de paginación se generó una variable global $sgte\_pagina\_libre$ que se inicializa en $0x100000$ siendo ésta la dirección a partir de la cual se pedirá memoria para las páginas. Luego se va pidiendo página libre con la función **get_next_free_page()** que devuelve el valor de la variable global e incrementa la misma en $0x1000$ (4k) para el siguiente llamado.

4.2. Ej4 b)

Se definió la rutina **mmu_inicializar_dir_zombi** que se encarga de inicializar un directorio y tabla de páginas para una tarea. Copia el código de la tarea de acuerdo a la posición en el mapa, y mapea esa página así como las 8 páginas de su alrededor (forman una matriz de 3x3) a la dirección virtual siempre ubicada a partir de la posición $0x800000$. y las 8 páginas siguientes
El código es el siguiente:

Algorithm 2 mmu_inicializar_dir_zombi(tipo, player,x,y)

Require: int tipo_zombi, int player, int x, int y
Ensure: retorna la posición de memoria pedida para el directorio de pagina

```
1: PageDirectory * PD ← casteo_a_puntero(get_next_free_page) //la sgte pag libre
2: PageTable * PT ← casteo_a_puntero(get_next_free_page) //la sgte pag libre
3: //inicializamos la primer entrada del PageDirectory
4: PD[0].dir_base ← dirTable >> 12
5: PD[0]. us/ rw / p ← 1 / 1 / 1
6: //todos los demás atributos de PD[0] se setean con cero
7: //y todas las demás entradas del PD se llenan con ceros (Bit P como no presente)
8: for i←1, hasta 1023 do
9:    PD[i] ← 0x00000000
10: end for
11: //luego seteamos las entradas de la PageTable (con identity_mapping)
12: unsigned int dir_fisica ← 0
13: for i←0, hasta 1023 do
14:    PT[i].dir_base ← dir_fisica
15:    PT[i].us / rw / p ← 1 / 1 / 1
16:    //todas los demás atributos de PT[i] se setean con cero
17:    dir_fisica ← dir_fisica +1
18: end for
19: mapear_pagina_a_mapa(x,y,dirPage,player,tipo_zombi)
20: return dirPage
```

La función **mapear_pagina_a_mapa** se encarga de mapear desde el área de memoria física en el mapa a la posicion indicada y luego de copiar el código de la tarea a la posición en el mapa.

4.3. Ej4 c)

se definen las funciones **mapear_pagina** y **unmapear_pagina**:

Algorithm 3 mapear_pagina

Require: memoria_virtual, cr3, memoria_fisica, id $U/S = \{1, 0\}$

```
1: PageDirectory *PD← castear_como_puntero(cr3)
2: index_pd ← obtener_PDindex(virtual)
3: index_pt ← obtener_PTindex(virtual)
4: if si la entrada del directorio tiene la tabla no presente then
5:    //se pide memoria para una pagina y se agrega
6:    PageTable *PT ← cateo_a_puntero(pido_pagina_libre())
7:    for Para i desde 0 hasta último índice de la tabla do
8:       seteamos todas las entradas en 0 (bit presente en cero)
9:       PT [i] ← 0x00000000
10:   end for
11:   PD[index_pd].dir_base ← pagina libre pedida
12:   PD[index_pd].us ← id
13:   PD[index_pd].rw ← 1 //lectura escritura
14:   PD[index_pd].p ← 1 //presente
15: else
16:   PageTable *PT ← (PD[index_pd].dir_base)<<12
17: end if
18: //por último se llena la entrada de la tabla de pagina
19: PT[index_pt].dir_base ← (memoria_fisica)>>12
20: PT[index_pt].rw ← 1
21: PT[index_pt].p ← 1
22: PT[index_pt].us ← id
```

Algorithm 4 unmapear_pagina

Require: memoria_virtual, cr3

```
1: PageDirectory *PD ← castear_como_puntero(cr3)
2: index_pd ← obtener_PDindex(virtual)
3: index_pt ← obtener_PTindex(virtual)
4: PageTable *PT← PD[index_pd].dir_base
5: //por último se setea todos los atributos con cero (bit P=0)
6: PT[index_pt] ← 0x00000000
```

5. Ejercicio 5

5.1. Ej5 a)

Se completaron 3 entradas necesarias de la **IDT** en el archivo $idt.c$ (IDT_ENTRY(32), IDT_ENTRY(33) y IDT_ENTRY_SYS(102)) para la interrupción del reloj, del teclado y de software respectivamente (102 es 0x66).

5.2. Ej5 b)

Escribimos la rutina asociada a la interrupción del reloj. Para esto usamos una función ya definida (**proximo_reloj**) definida en $isr.asm$:

```
_isr32:
       pushad
        call fin_intr_pic1
        call proximo_reloj
        popad
        iret
```

5.3. Ej5 c)

Escribimos la rutina de atención para el teclado asociada a la interrupción 33 en $isr.asm$:

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

Donde **atención_teclado** está definida en $idt.c$ de la siguiente manera:

```
void atencion_teclado(unsigned int t)
{
    switch(t){
    case 0x1e: //A
            print_hex(t,3,75,0,C_FG_WHITE);
            break;
    case 0x11: //W
            print_hex(t,3,75,0,C_FG_WHITE);
            break;
    ...
    //asi con todas las entradas (s,d,i,k,l,j,Lshift,Rshift,y)
    ...
    default:
            break;
    }
}
```

5.4. Ej5 d)

Por ultimo se pide agregar la rutina para la interrupción $0x66$

```
;; Rutinas de atención de las SYSCALLS
;; ------------------------------------------------------------------------ ;;
   _isr102:
    mov eax,0x42
    iret
```

6. Ejercicio 6

6.1. Ej6 a)

En el archivo **gdt.c** se crearon 18 nuevas entradas a la GDT para las tareas: inicial, idle, 8 tareas para jugador A y 8 para jugador B, quedando la GDT definida de la siguiente manera:

Descripcion	Index	Base	Limite	Tipo	S	P	DPL	AVL	L	D/B	G
Nulo	00	0x0	0x0	0x0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...
pantalla	12	0xb8000	0x5fff	0x03	1	1	3	0	0	1	0
T_inicial	13	0x000	0x0068	0x09	0	1	0	0	0	1	0
T_idle	14	0x000	0x0068	0x09	0	1	0	0	0	0	0
T_1A	15	0x000	0x0068	0x09	0	1	0	0	0	0	0
T_2A	16	0x000	0x0068	0x09	0	1	0	0	0	0	0
T_iA i-(3..7)	...	idem ant	...	...	...	...	...	...	...	...	...
T_8A	22	0x000	0x0068	0x09	0	1	0	0	0	0	0
T_1B	23	0x000	0x0068	0x09	0	1	0	0	0	0	0
T_iB i-(2..7)	...	idem ant	...	...	...	...	...	...	...	...	...
T_8B	30	0x000	0x0068	0x09	0	1	0	0	0	0	0

6.2. Ej6 b,d,e)

En el archivo **tss.c** se inicializa la tss de la tarea idle, así como también se coloca de manera dinámica la dirección base en el descriptor de la GDT correspondientes a la tarea Idle y tarea inicial.

```
void inicializar_idle(){
  tss_idle.eip = 0x16000;
  tss_idle.ebp = 0x27000;
  tss_idle.esp = 0x27000;
  tss_idle.es = 0x060;
  tss_idle.cs = 0x40;
  tss_idle.ss = 0x48;
  tss_idle.ds = 0x48;
  tss_idle.fs = 0x48;
  tss_idle.gs = 0x48;
  tss_idle.cr3 = 0x27000; //el mismo cr3 que el kernel
  tss_idle.eflags = 0x202;  //con interrupciones habilitadas
  //llenamos la direccion base del descriptor  tss de tarea inicial
    gdt[tarea_inicial].base_0_15 = (unsigned short) ((unsigned int) &tss_inicial);
    gdt[tarea_inicial].base_23_16 = (unsigned char) ((unsigned int) &tss_inicial)>>16;
    gdt[tarea_inicial].base_31_24 = (unsigned char) ((unsigned int) &tss_inicial)>>24;
//llenamos la direccion base del descriptor tss de tarea idle
```

```
    gdt[tarea_idle].base_0_15 = (unsigned short) ((unsigned int) &tss_idle);
    gdt[tarea_idle].base_23_16 = (unsigned char) ((unsigned int) &tss_idle)>>16;
    gdt[tarea_idle].base_31_24 = (unsigned char) ((unsigned int) &tss_idle)>>24;
}
```

6.3. Ej6 c)

Las **TSSś** con los datos correspondientes para las tareas $ZOMBIS$ se llenan dinámicamente en el archivo **tss.c**. Dentro de la función $tss\_zombi\_lanzar()$ se encuentra el código que hace lo siguiente:

```
...
if(jugador == 0)
{
tarea_zombi = info_zA; //arreglo de estructura con atributos para zombi
tarea_tss = tss_zombisA;//arreglo de TSSs de tarea A
}
else
{
tarea_zombi = info_zB; //arreglo de estructura con atributos para zombi
tarea_tss = tss_zombisB; //arreglo de TSSs de tarea A
}
    ...
    ...
tarea_tss[i].eip = 0x8000000;
tarea_tss[i].ebp = 0x8001000;
tarea_tss[i].esp = 0x8001000;
tarea_tss[i].es = (((unsigned int) user_data) << 3) + 0x3; //descriptores de segmento de datos lvl 3
tarea_tss[i].cs = (((unsigned int) user_code) << 3) + 0x3; //descriptores de segmento de codigo lvl 3
tarea_tss[i].ss = (((unsigned int) user_data) << 3) + 0x3;
tarea_tss[i].ds = (((unsigned int) user_data) << 3) + 0x3;
tarea_tss[i].fs = (((unsigned int) user_data) << 3) + 0x3;
tarea_tss[i].gs = (((unsigned int) user_data) << 3) + 0x3; //
tarea_tss[i].cr3 = mmu_inicializar_dir_zombi(t,jugador,x,y);
tarea_tss[i].eflags = 0x202;
tarea_tss[i].esp0 = get_next_free_page() + 0x1000;
tarea_tss[i].ss0 = (unsigned int) (kernel_data << 3);
    ...
```

Nota: En el código las contastantes $kernel\_code$ (8), $kernel\_data$ (9), $user\_code$(10) y $user\_data$(11) corresponden a los índices en la GDT para cada descriptor. Por ejmplo $kernel\_code$ es el índice de la GDT para el segmento de código del kernel.

6.4. Ej6 f)

Para saltar de la tarea inicial a la trea Idle se hace lo siguienteb en elarchivo **kernel.asm**

```
; Cargar tarea inicial
    mov ax, 0x68
    ltr ax
    jmp 0x70:0
```

7. Ejercicio 7

7.1. Ej7 a)

Se tiene 3 estructutas importantes definidas en en **tss.h**:

- **tss** que guarda el contexto de todas las tareas. De esta estrutura se tienen 2 (una para la tarea inicial y otra para la idle); y 2 vectores de 8 Tss cada una: un vector para guardar las tareas del jugador A y otro para las tareas del jugador B indexadas por supuesto por el nro de tarea.
- **info_tss** que contiene informacion de la tarea/zombie, como posicion, tipo, etc. Se tienen 2 vectores de 8 info_tss cada uno. Que sirve para guardar la información de las tareas en ejecución de cada jugador.
- **player** que es la estructura que actua como jugador en el juego, guarda información como la altura en el mapa y el tipo de zombi antes de lanzarce, la cantidad de zombis disponible, puntos, etc. Se tiene un vector con 2 player, uno por cada jugador.

Para iniciar el scheduler se llama a la siguiente función definida en **sched.c**

```
void iniciar_sched()
{
    players();
//players() setea los valoresiniciales para las estr. player e info_tss
    unsigned int a1 = altura_player(0);
    unsigned int a2 = altura_player(1);
    print("M",0,a1+1,C_FG_WHITE);
    print("M",79,a2+1,C_FG_WHITE);
//luego se muestra en pantalla las posiciones iniciales de los jugadores
}
```

7.2. Ej7 b)

Se pide definir la función **sched_proximo_indice** encargada de decidir cual será la siguiente tarea a ejecutarse. Como lo pide el enunciado los zombis de cada uno de los jugadores representarán 2 conjuntos distintos (uno por cada jugador), y por cada tick de reloj se ejecutará un zombi de cada jugador (de cada conjunto) por vez. Dentro de cada conjunto los zombis también tienen definido un orden y es cíclico.
Esta función también debe asegurarse de que la tarea Idle sea la que se ejecute cuando y no hayan más tareas que correr, o cuando una tarea es interrumpida y no haya terminado su quantum (en este caso esta porción de tiempo restante será ocupada por la Idle).

```
unsigned short sched_proximo_indice() {
  unsigned short next = 0x00;
  if(!r_pause() && !hay_ganador() && !sin_zombis())
  //si no esta pausado, no hay ganador y hay zombis
  {
    if(j_actual == 0)//si jug actual es 0
    {
      next = prox_Z_tarea(1) + 0x00;
      j_actual = 1;//cambio de jugador
    }
    else if(j_actual == 1)
    {//idem caso anterior
      next = prox_Z_tarea(0) + 0x00;
      j_actual = 0;
    }
  }
  return next;//se devuelve el indice en la gdt de proxima tarea
```

La función **prox_Z_tarea** está definida en **tss.c** y se encarga, dado un jugador, buscar entre su conjunto de tareas activas el selector de la tss en la GDT del siguiente zombi, o el selector de la idle en la GDT si no lo hubiese. La descripción del funcionamiento de la función es la siguiente:

- Se busca entre las 8 posiciones y a partir de la posición de la última que ejecutó, la siguiente que contenga una tarea activa. Pueden ocurrir 2 cosas.
  - si encuentra la siguiente prepara el selector para devolverla, ajusta las estructuras mencionadas al comienzo del ejercicio 7 para esa tarea que ejecutará. Imprime en pantalla el caracter que representa su reloj (caracter debajo del slot en pantalla).
  - Si no encuentra ninguna tarea prepara el selector de la tarea Idle para devolver.
- Si la cantidad de jugadores activos es la máxima (8) y ya pasaron activos los 8 durante un tiempo determinado, se da por terminado el juego y se imprime el ganador.
- Se devuelve el selector de la siguiente tarea a ejecutar

7.3. Ej7 c)

Se modificó la rutina de la interrupción $0x66$ y se implementó el servivio *mover*. En el archivo $isr.asm$ donde se define el handler de atención a este servicio (_isr102:) siempre que en eax se le pase una codificación de un movimiento válido (0x83D, 0x732, 0x441 o 0xAAA) hace un llamado a la función $mover\_zombi$ definida en $idt.c$ de la siguiente manera:

```
void mover_zombi(unsigned int cr3, unsigned int mov)
{
    info_tss* zombi <- info_tss de tarea en ejecucion;
    int j <- jugador actual;
    int x <- pos_x de tarea;
    int y <- pos_y de tarea;
    char c <- tipo_zombie;
    //dependiendo si 'mov' es alguna de las cuatro codificaciones se llama a:
    mov_adelante(cr3,x,y,j,c);
    ó
    mov_derecha(cr3,x,y,j,c);
    ó
    mov_izquierda(cr3,x,y,j,c);
    ó
    mov_atras(cr3,x,y,j,c);
}
```

Cada una de de las cuatro funciones mov_xxxx realiza el movimiento pedido dependiendo del jugador. Por ejemplo $mov\_adelante$ tiene el sgte comportamiento:

- Si se llegó hacia el borde contrario de donde empezó se desaloj la tarea, y suma puntos el jugador contrario.
- Como se sabe, cada posición que recorre el zombi en la pantalla representa una página física en la memoria (una al lado de la otra a partir de la posición 0x400000 física). Al mover hacia adelante, se hace el remapeo en la unidad de paginación hacia la pagina contigua (la siguiente si es el jugadorA o la anterior si es el jugadorB)y no solo de esa página sino también de las 8 páginas representadas en pantalla que bordean la posición del zombi.
- Luego se copia el código del área de kernel a la página recién mapeada.

Los otros tres movimientos son análogos al definido arriba con la salvedad que para el jugadorB sería siempre la dirección contraria (viendola en el mapa) que para el jugadorB. Los movimientos hacia arriba y hacia abajo son cíclicos, por ejemplo si se llega al extremo superior de la pantalla, y se quiere mover un casillero más arriba, se retorna en la misma columna pero en la fila inferior.

7.4. Ej7 d)

En este ejercicio se tuvo que modificar la rutina de atencion de reloj de modo que quedara de la siguiente manera.

```
_isr32:
  pushad
      call proximo_reloj
      call sched_proximo_indice
      cmp ax, 0
      je .no_jump
      ;xchg bx, bx
      mov [sched_tarea_selector], ax
      call fin_intr_pic1
      jmp far [sched_tarea_offset]
      jmp .sched_end
      .no_jump:
      call fin_intr_pic1
  .sched_end:
  popad
iret
```

De esta manera en cada ciclo de reloj, se produce correctamente el intercambio de tareas. El codigo de esta rutina fue provista por la catedra, y gracias a la implementacion de la funcion sched_proximo_indice() no hubo que hacer ningun cambio adicional en esta rutina, ya que la funcion del scheduler retorna correctamente la proxima tarea a ser intercambiada.

7.5. Ej7 e

Para resolver una excepción se define la función **resolver_excepcion()** que es llamada desde $isr.asm$. La función básicamente imprime una leyenda con la descripción de la misma. Ejemplo "$DivideError$", "$Overflow$", etc. Luego se llama a **desalojar_tarea()** pasándole como parámetro el selector de la misma. La función está definida en $tss.c$ hace lo siguiente:

- busca en la estructura del tss_info de la tarea y setea los atributos para indicar que no esta activa.
- Con respecto al jugador le resta uno a la cantidad de zombis en juego. Marca en el slot inferior de la pantalla con una X para indicar que el zombi fue sacado de ejecución.
- Si el jugador se quedó sin mas zombis para lanzar y sin zombis en el juego, se muestra el cartel con el ganador, o con el empate si así fuera el resultado.

7.6. Ej7 f)

Para hacer este ejercicio se tuvo que agregar las siguientes variables y funciones definidas en **tss.c**:

- **pause**: Char que inicia en 0 como pausa no habilitada.
- **debug_y**: inicia en 0 para comportarse como un debug off.
- Para intercambiar valores de estas variables usamos las siguientes funciones:
- **r_pause()**: Retorna la variable de pausa en 1 si lo esta, y 0 si no. Se usa como función booleana.
- **set_pause()**: Setea la variable de pause en 0 o 1.
- **debug_on()**: Retorna el 1 si esta activo el debug, y 0 si no.
- **press_debug()**: Itercambia el valor de debug en 0 (OFF), y 1 (ON).

Los valores para estas variables se cambian o verifican cuando atiende la interrupcion de teclado para la tecla 'y' en la funcion **atencion_teclado**. El codigo que se ejecuta en esa interrupcion es el siguiente:

```
//codigo:
press_debug(); //habilito o desahilito el debug
print_intr_line(); //funcion implementada en screen.h
if(r_pause()) // si estaba habilitada la pausa, entonces el valor de pause es 1, lo cual
{   //tambien significa que se capturo una excepcion y
      //se imprimio el estado del procesador, y el juego esta pausado
      print_back(); //limpia el sector que muestra el estado del procesador
      set_pause(); // cambia el valor de pause por 0, para deshabilitar la pausa del juego
}
```

print_back está definida en **screen.h**. Para que esta función pinte de nuevo el estado anterior de la pantalla se agregó en **screen.c** una matriz con las mismas dimensiones que la pantalla de estado, en la que se guardó los datos de atributos de colores de cada pixel del mapa antes de que se pinte la pantalla de estado.

Si se activa el debug, si una tarea produce una excepción, al producirse esta misma se tiene que imprimir el contexto actual de los registros. Como ya vimos en el item anterior la función **resolver_excepcion** identifica que interrupción se produjo y la imprime en la parte superior de la pantalla. Para que también pueda imprimir la pantalla de estado, se tuvo agregar mas parámetros a la aridad de la funcion, los cuales representan todos los registros del contexto actual de la tarea que se piden a imprimir. La aridad de la funcion queda de la siguiente manera:

void resolver_excepcion(int i,int ebx,int ecx.../* –todos los registros que se piden– */)

De esta manera se puede llamar a la funcion **print_debug**, declarada en **screen.h**, que toma todos los parametros correspondientes a los registros para que se puede imprimir los valores se pide que se muestre en la pantalla de estado.
Una vez hecho todo esto se habilita la pausa del juego (**set_pause**), para detener la ejecución de las tareas que en ese momento están corriendo.
También se pide que se desaloje la tarea actual que produjo la excepción, por lo tanto se llama a **desalojar_tarea**. Dado que **desalojar_tarea** también imprime en la pantalla una 'x' en la última posición que ocupó la tarea, se debe llamar antes que a **print_debug** para que no imprima algo indeseado en la pantalla de estado.

Como es el macro ISR que llama a una interrupción de sistema, es el que llama a **resolver_excepcion**, y esta toma mas parametros que antes se tienen que pushear. El cambio fue el siguiente:

```
_isr%1:
    push eax
    mov eax, [esp + 12]  ; eip   o es correcto desplazarme 12 o en verdad es 8
    push eax
    mov eax, [esp + 20] ; eflags
    push eax
    xor eax, eax
    mov ax, ss
    push eax
    ;se pushean todos los registros selectores de la misma manera
    ...
    ...
    mov eax, cr4
    push eax
    ;se pushea el cr4, cr3, cr2, cr0
    ...
    push esp
```

```
mov eax, [ebp]   ;pusheo los valores de la pila/stack
push eax
mov eax, [ebp+4]            ;si es que estoy leyendo en el lugar adecuado
push eax
mov eax, [ebp+8]
push eax
mov eax, [ebp+12]
push eax
mov eax, [ebp+16]
push eax
;pusheo ebp, edi, esi, edx, ecx, ebx
...
mov eax, %1
push eax
call resolver_excepcion
jmp 0x70:0
```

De esta manera la pantalla de imprime el contexto actual de los registros del procesador. Al finalizar, se salta la idle.
NOTA: Tal vez no era necesario mostrar esta parte del código pero, nos gustaria que se tome en cuenta las líneas comentarios, ya que al probar el funcionamiento del debugger, no se está seguro si se leyó correctamente los valores de stack que se pedía, y el valor de eip.