DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Alejandro Albertini	924/12	ale.dc@hotmail.com
Tomas Shaurli	671/10	zeratulzero@hotmail.com
Sebastian Aversa	379/11	sebastianaversa@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

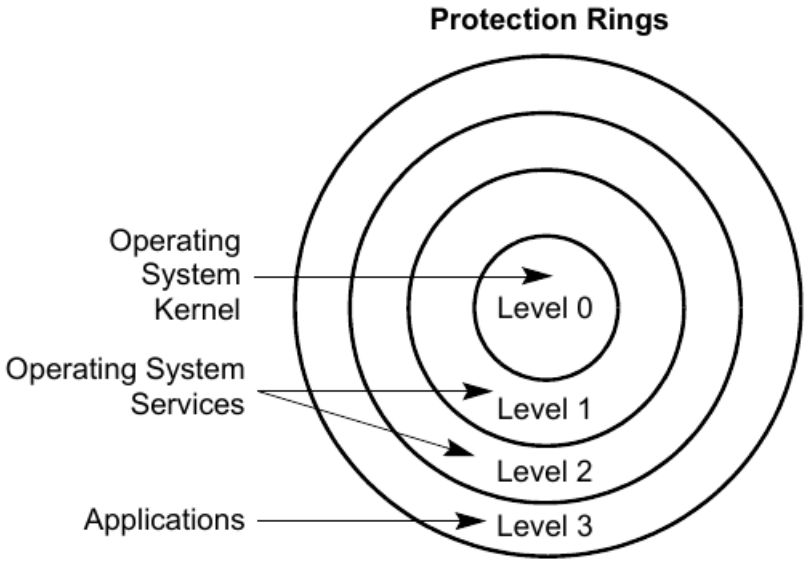

Figura 1: Niveles de Privilegio.

1. Introducción

En este trabajo práctico implementamos un sistema operativo de 32 bits que corre en Modo Protegido. Este modo permite, entre otras mejoras respecto al Modo Real, realizar *multitasking*–es decir, correr tareas concurrentemente, en instantes de tiempo que se solapan entre si, y dando la impresión al usuario de que se están ejecutando todas al mismo tiempo. Esto se realiza de una forma segura, controlado por un sistema de privilegios, y transparente. Además, este modo permite direccionar hasta 4GB de memoria e implementar memoria virtual, permitiendo a una aplicación correr en su propio espacio aislado de memoria y la posibilidad de utilizar una cantidad de memoria mayor a la que se dispone físicamente.

2. Niveles de privilegio

El Modo Protegido permite contar con hasta 4 niveles de privilegio numerados desde 0 hasta 3, siendo 0 el de mayor privilegio y 3 el de menor. (Figura 1).

En nuestro sistema empleamos sólo 2 de estos niveles: el nivel 0, al que también llamaremos *modo supervisor* o *modo kernel*, y nivel 3, también llamado *modo tarea* o *modo usuario*. El *kernel* corre en nivel 0 y el resto de las tareas en nivel 3. La única excepción es la tarea `Idle`, que cuenta con nivel de privilegio 0.

3. Memoria

3.1. Segmentación

Una funcionalidad del modo protegido es la segmentación de la memoria. Esto permite particionar la memoria en segmentos de distintas características, logrando crear espacios de direcciones separados y aislados entre sí. Dichos segmentos se pueden definir en cualquier lugar de memoria física, y su tamaño puede ser variable. Además de un nivel de privilegio asignado, los mismos tienen asociado un tipo. Ese tipo puede determinar, por ejemplo, que un segmento se utilizará para código o para datos, o que será de sólo lectura, brindando protección al sistema. Con este sistema se obtiene un espacio de direccionamiento *lineal*, donde cada dirección lineal de memoria consiste en un selector de segmento y un offset dentro de él (Sel:Offset).

Índice	Base	Límite	Tipo	Privilegio
9	0x0	0x2DD00	Datos	0
10	0x0	0x2DD00	Datos	3
11	0x0	0x2DD00	Código	0
12	0x0	0x2DD00	Código	3
13	0xB8000	0x2FA00	Datos	0

Figura 2: Definiciones de los segmentos en la `GDT`.

En todos los casos, para utilizar el Modo Protegido se debe definir al menos un segmento, ya que no es posible desactivar el mecanismo de segmentación. Para eso se deben crear tantas entradas como segmentos se quieran definir en un espacio reservado en memoria reservado **Global Descriptor Table (GDT)**. El comienzo de dicho espacio deberá estar alineado a 8 bytes en memoria y su dirección deberá estar cargada en el registro **Global Descriptor Table Register (GDTR)** para que el procesador pueda acceder a la información contenida en la tabla.

Nuestro sistema cuenta con cuatro segmentos principales que cubren los primeros 623MB de memoria física. Dos de ellos son de código, uno definido como nivel 0 y otro como nivel 3, y otros dos de datos, también definidos para cada nivel de privilegio. Esta configuración es conocida como *Modo Flat Protegido*. Los segmentos brindan protección, pero no hay una separación a nivel de espacio de direcciones; todos los segmentos se mapean a las mismas secciones de memoria física.

Otro segmento con el que cuenta nuestro sistema cubre la porción de memoria de video correspondiente a la pantalla. El mismo está definido con nivel de privilegio 0.

Las definiciones de los segmentos de nuestro sistema pueden verse en la Figura 2.

Para acceder a cada uno de los segmentos, el procesador cuenta con registros de segmento. Los mismos son los registros **CS** (segmento de código); **DS** (segmento de datos); **SS** (segmento de pila); y **ES**, **FS** y **GS** (segmentos de propósito general) que almacenan el índice dentro de la **GDT** del segmento al que seleccionan.

3.2. Paginación

Este mecanismo permite que una aplicación utilice todo el espacio lineal de direcciones sin que sea necesario contar con toda esa cantidad de memoria física. Además brinda protección adicional a la segmentación, aislando las tareas y permitiendo que sólo lean y escriban en posiciones de memoria que se encuentran mapeadas a su espacio privado de direcciones. Este modo se puede activar o desactivar a través del registro **CR0**.

Nuestro sistema utiliza paginación, para lo cual define todas las estructuras necesarias (directorios y tablas de página). El kernel y cada tarea que corre en el sistema cuentan con su propio conjunto de directorio y tablas. Como consecuencia, cada uno posee su propio valor dentro del registro **CR3**, el cual apunta al Directorio de Tablas de Página (*Page Directory Table*) que el procesador utilizará para traducir direcciones de memoria lineales a físicas.

Para el kernel y las tareas, las direcciones desde 0x0 hasta 0xDC3FFF (que contienen todo el espacio del kernel, el área libre y *el mapa*) se mapean con *identity mapping*, es decir que las direcciones lineales en ese rango se mapean a la misma posición dentro de la memoria física. A diferencia del kernel, las tareas además tienen dos páginas de direcciones virtuales (con permisos de lectura/escritura y nivel de privilegio 3) mapeadas a direcciones físicas dentro de *el mapa* y pueden modificar su espacio de memoria dinámicamente, adicionando direcciones virtuales (también mapeadas a posiciones dentro de *el mapa*) a su espacio de direccionamiento. Para eso el sistema cuenta con mecanismos para modificar el directorio y tablas de una tarea dada y mapear direcciones virtuales a físicas. Entre ellos las funciones `void mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned char attrs)` y `void mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`. El mapa de memoria puede verse en detalle en la Figura **??**.

La sección del kernel que administra la memoria se denomina **MMU (Memory Management Unit)** y su estructura cuenta con los siguientes componentes:

- Las variables:

  `mmu_entry* page_dir`
  : Ahí empieza el directorio de paginas del kernel.

  `mmu_entry* page_libre`
  : A partir de acá pido paginas libres cada vez que necesite.

  Todas se incrementan de a 4 KB o 0x1000.

- Las funciones:

  `mmu_inicializar()`
  : Inicializa la unidad de administración de memoria del kernel.

  `void mmu_inicializar_dir_kernel`
  : Crean y completan el directorio y las tablas de página del kernel requeridas para realizar identity mapping para mapear los primeros 623 MB de memoria.

  `mmu_entry* mmu_inicializar_dir_zombie`
  : Crean y completan el directorio y las tablas de página del kernel para un zombie.

  `mmu_entry* area_libre_mmu`
  : Apunta a una nueva pagina libre del área de memoria libre.

  `void mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned cha`
  : Permite mapear una dirección virtual a una física dentro de un directorio dado con los atributos pasados por parámetro.

  `void mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`
  : Realiza el proceso inverso.

  `void copiar_pagina_4k(unsigned int* dst, unsigned int* src)`
  : Copia todo el contenido de una pagina de 4KB.

  `int primer_pos_libre(int* tareas)`
  : Devuelve la primer posicion libre del vector.

4. Modo Protegido

Como dijimos anteriormente, el modo protegido presenta mejoras significativas con respecto al modo real. Sin embargo, por una cuestión de compatibilidad hacia atrás, los procesadores de la arquitectura IA-32 arrancan en modo real. El cambio de modo se debe realizar manualmente y no puede hacerse hasta no tener definida una mínima estructura (una GDT con al menos un segmento, como vimos en la sección 3.1).

El Código 1 muestra las instrucciones que ejecuta nuestro kernel desde que inicia hasta que pasa a modo protegido.

```
BITS 16
start:
    ; Deshabilitar interrupciones
    cli

    ; Cambiar modo de video a 80 X 50
    mov ax, 0003h
    int 10h ; set mode 03h
    xor bx, bx
    mov ax, 1112h
    int 10h ; load 8x8 font

    ; Imprimir mensaje de bienvenida
    imprimir_texto_mr iniciando_mr_msg, iniciando_mr_len, 0x07, 0, 0
```

```
    ; 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:
    ; Estamos en modo protegido.
```

Código 1: Pasaje a modo protegido – **kernel.asm**

Se puede observar como, además de cambiar el modo de video y de imprimir un mensaje de bienvenida, se llama a la rutina `habilitar_A20`. La compuerta A20 es un rezago histórico que se implementó improvisadamente para mantener compatibilidad con sistemas viejos y que sigue existiendo hasta el día de hoy. Es una compuerta lógica que habilita o deshabilita la línea 20 del bus de direcciones del procesador (A20). Como en modo protegido interesa direccionar más memoria que el límite de modo real de 1 MB, el primer paso antes de poder hacer el salto entre modos es habilitar dicha compuerta.

El siguiente paso es cargar el registro `GDTR` con los datos necesarios para que el procesador encuentre nuestra **GDT**. Ese dato no es sólo la dirección, sino también el tamaño de dicha tabla. Ese par de datos se denomina *descriptor* y su estructura puede verse en el Código 2. Para cargar dicho registro se utiliza la función específica `lgdt` (Load GDTR Register).

Luego se activa el bit 0 (**PE**: Protection Enable) del registro **CR0** (Figura 3). Una vez hecho esto es muy importante ejecutar el salto que se encuentra en la línea de abajo y no otra instrucción: En ese punto el procesador ya activó el modo protegido y tiene cargados los datos de la **GDT**. Sin embargo, el contenido del registro **CS** al arranque (el cual selecciona el segmento de código desde donde se leen las instrucciones a ejecutar) es 0. Por diseño, es ilegal definir un segmento en el índice 0 de la **GDT** y que los registros de segmento **CS**, **DS** o **SS** apunten al índice nulo. Además, no existe ninguna instrucción que permita modificar el registro **CS**. Ese tipo de salto (denominado *far jump*) es la única manera de modificar el registro **CS**, en este caso seleccionando el segmento de código perteneciente al kernel. La única forma de hacer el pasaje de modo es si ese salto se realiza justo después de setear el bit **PE**, mientras el **Instruction Pointer (EIP)** apunta a la posición de memoria donde se encuentra la instrucción. Cualquier otra instrucción resultaría en una excepción de *protección general* (#GP).

```
typedef struct str_gdt_descriptor {
    unsigned short  gdt_length;
    unsigned int    gdt_addr;
} __attribute__((__packed__)) gdt_descriptor;

gdt_descriptor GDT_DESC = {
    sizeof(gdt) - 1,
    (unsigned int) &gdt
};
```

Código 2: Descriptor de GDT – **gdt.{c,h}**

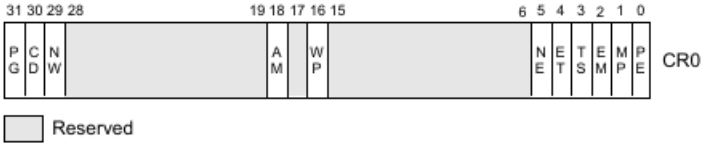

Figura 3: **Control Register 0** (El bit 0 activa el modo protegido).

5. Interrupciones

Nuestro sistema soporta 2 interrupciones por hardware: las producidas por el timer y las generadas por el teclado. Además cuenta con rutinas de atención a 18 *excepciones* (interrupciones generadas por el procesador cuando encuentra un error en la ejecución de una instrucción) y una *syscall*, numerada 0x66, que implementa funcionalidades del juego y a la que las tareas pueden llamar.

5.1. Estructuras

La arquitectura IA-32 cuenta con una estructura que facilita la administración de la atención a las interrupciones. Similar a la **GDT**, la **Interrupt Descriptor Table (IDT)** cuenta con entradas secuenciales de descriptores, generando un vector donde cada índice dentro de dicha tabla corresponde a una interrupción. Se pueden definir hasta 255 interrupciones distintas y aquellas entradas que no se deseen utilizar pueden quedar nulas. Al igual que con la **GDT**, hay un registro específico que apunta al comienzo de la tabla: el registro **IDTR**. En nuestro sistema la IDT es la misma tanto para el kernel como para las tareas.

Para cargar el registro **IDTR** se cuenta con la instrucción *lidt* que toma como parámetro un *descriptor* de IDT. (Figura 3)

```
typedef struct str_idt_descriptor {
    unsigned short  idt_length;
    unsigned int    idt_addr;
} __attribute__((__packed__)) idt_descriptor;

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

Código 3: Descriptor de IDT – **idt.{c,h}**

Cada entrada en la **IDT** permite asignar un nivel de privilegio a cada rutina de atención a esa interrupción. Todas nuestras rutinas están definidas con nivel de privilegio 0, excepto por la rutina de atención a la syscall 0x66 que cuenta con nivel de privilegio 3, ya que debe poder ser ejecutada por las tareas sin cambiar de privilegio.

La arquitectura permite utilizar distintos tipos de descriptores en la **IDT**. Nuestro sistema utiliza *interrupt-gate descriptors*, los cuales permiten llamar directamente al código en Assembler que atiende a la interrupción.

Una vez cargada la IDT (`lidt [IDT_DESC]` en el archivo **kernel.asm**), y luego de haber inicializado el sistema en su totalidad, se activan las interrupciones mediante la instrucción `sti`, que setea el bit 9 (**IF**) del registro **EFLAGS**.

5.2. Excepciones

Como las tareas pueden tener cualquier tipo de comportamiento, el sistema posee un mecanismo que permite resolver cualquier problema que las mismas generen. La acción que realiza dicho mecanismo es simple: desaloja a la tarea infractora y deja el sistema listo para seguir ejecutando las demás. Esa política

Cuadro 1: Entradas en la **IDT**

Vector Nro.	Descripción
0	#DE: Divide Error
1	#DB: Reserved
2	NMI: Nonmaskable External Interrupt
3	#BP: Breakpoint
4	#OF: Overflow
5	#BR: BOUND Range Exceeded
6	#UD: Invalid (Undefined) Opcode
7	#NM: Device Not Available
8	#DF: Double Fault
10	#TS: Invalid TSS
11	#NP: Segment Not Present
12	#SS: Stack-Segment Fault
13	#GP: General Protection
14	#PF: Page Fault
16	#MF: x87 FPU Floating-Point Error
17	#AC: Alignment Check
18	#MC: Machine Check
19	#XM: SIMD Floating-Point Exception
32	Reloj
33	Teclado
82	Syscall 0x52 (DPL: 3)

es independiente de la causa que haya generado los inconvenientes, por lo que el código que se utiliza para atender las excepciones es idéntico en cada una.

El caso donde la tarea `Idle` produce una excepción es un evento fatal en nuestro sistema y por lo tanto no podemos continuar operando.

5.3. PIC

El **Controlador Programable de Interrupciones (PIC)** es un controlador que permite administrar señales que provienen desde dispositivos de hardware que desean capturar la atención del procesador. Para eso utiliza un sistema de prioridades y permite disponer de una menor cantidad de líneas de las que serían necesarias si todos los dispositivos se conectaran directamente con el procesador. Al ser programable, permite mapear eventos externos a distintas interrupciones e incluso pueden conectarse varios controladores en cascada para aumentar la cantidad de señales que un procesador puede recibir.

El PIC permite recibir 15 señales. Las mismas se encuentran mapeadas por defecto a interrupciones que ya están designadas por el procesador como excepciones. Por esa razón, antes de poder habilitar las interrupciones, es necesario remapear dichas señales a indices del vector de interrupciones que estén disponibles. Para hacer esto, utilizamos las funciones `deshabilitar_pic`, `resetear_pic` y `habilitar_pic` que nos fueron provistas por la cátedra, y que mapean la interrupción del **timer** al índice **0x32** y la del **teclado** al **0x33**.

El PIC cuenta con un registro llamado **ISR (In-Service Register)** que permite determinar si una interrupción particular está siendo atendida. En el modo de operación en el que utilizamos el PIC en este TP es necesario avisarle al controlador que ya atendimos a dicha interrupción. Para eso utilizamos la función `fin_intr_pic1` provista por la cátedra, cuyo uso puede observarse en las dos rutinas detalladas más abajo.

5.4. Teclado

El sistema soporta la entrada desde el teclado. Las teclas soportadas son las **W y S** para mover al jugador A, **I y K** para mover al jugador B, **A y D** para cambiar al zombie del jugador A, **J y L** para cambiar al zombie del jugador B, **Lshift** para lanzar un zombie del jugador A y **Rshift** para lanzar un zombie del jugador B.

```
1  _isr33:
2      pushad
3      cli
4      xor eax, eax
5      in al, 0x60
6      mov edi, eax
7
8      cmp edi, 0x15
9      jne .noEsDebugger
10
11     cmp byte [debug_activado], 1
12     je .restablecer_pantalla
13
14     xor eax, eax
15     mov al, byte[modo_debug]
16     xor al, 0x1
17     mov byte[modo_debug], al
18
19     jmp .fin
20
21 .restablecer_pantalla:
22     mov dword [entro_excepcion], 0
23     mov dword [debug_activado], 0
24     call restablecer_pantalla
25     jmp .fin
26
27 .noEsDebugger:
28     cmp byte [debug_activado], 1
29     je .fin
30
31     push edi
32     call print_keyboar_int
33     pop edi
34
35     push edi
36     call setear_tipo_zombie
37     pop edi
38
39     push edi
40     ;xchg bx, bx
41     call crear_zombie
42     pop edi
43
44     push edi
45     call mover_jugador
46     pop edi
47
48 .fin:
49     push dword[modo_debug]
50     call print_debug_mode
51     add esp, 4
52
```

```
53      call fin_intr_pic1
54      popad
55      iret
```

Código 4: Rutina de atención a la interrupción del teclado. – **isr.asm**

5.5. Timer

El timer gobierna el cambio en la ejecución de las tareas. Cada interrupción del timer le indica al sistema que debe dejar de ejecutar la tarea actual para pasar a ejecutar la siguiente. Se genera una interrupción cada vez que se alcanza una cantidad determinada de ciclos de reloj.

```
_isr32:
    pushad
    cli
    cmp byte [debug_activado], 1
    je .nojump

    call proximo_reloj
    call sched_proximo_indice ; ME DEVUELVE EN EAX EL SELECTOR DE LA GDT DE LA TAREA A SALTAR
    cmp ax, 8
    je .nojump

    mov [selector], ax
    call fin_intr_pic1
    jmp far [offset]
    jmp .end
    .nojump:
    call fin_intr_pic1

    .end:
    popad
    iret
```

Código 5: Rutina de atención a la interrupción del timer. – **isr.asm**

6. Tareas

Las dieciséis tareas que corren en el sistema lo hacen concurrentemente. Cada una posee un tiempo fijo (denominado **quantum**) dentro del cual pueden ejecutar su propio código. Una vez que ese tiempo se termina se pasa a ejecutar otra tarea durante un quantum, se avanza a la siguiente y así sucesivamente. Como se detalla en la sección anterior, un quantum de nuestro sistema equivale a un *tick* de reloj.

Para que ese proceso sea transparente a las tareas se debe de alguna forma preservar el contexto de las mismas. Esto es, suponiendo que una tarea se está ejecutando y tiene almacenado un valor en el registro `EAX` que espera utilizar más adelante, cuando su quantum finalice y su ejecución sea interrumpida, dicho valor debe ser almacenado en algún lado. De esa forma, cuando se desee continuar con la ejecución de la primera tarea durante otro quantum, la misma podrá seguir ejecutando sus instrucciones como si nada hubiera pasado en el medio y la misma tuviera el procesador a su disposición.

Para asistir con el proceso de guardar y restablecer contextos, la arquitectura cuenta con un mecanismo basado en **Task-State Segments (TSS)**, el cual utilizamos en nuestro sistema. Los **TSS** son porciones de memoria que se definen como índices en la **GDT**. Cada tarea tiene un registro, el **Task Register (TR)**, que selecciona un **TSS**. Cuando se realiza un cambio de tarea, el procesador automáticamente guarda el contexto de la tarea actual dentro del **TSS** seleccionado con el registro **TR** de esa tarea y selecciona un nuevo **TSS** para la nueva tarea, donde se almacenará su contexto al realizar un nuevo salto a otra tarea.

6.1. Task State Segments (TSS)

Hay dieciséis segmentos **TSS** definidos en la **GDT** de nuestro sistema y además `GDT_IDX_TSS_INICIAL`. (Esta última se utiliza sólo una vez: al inicio del sistema, cuando se realiza el salto a la primera tarea, la `Idle`.) Después se comienza ejecutando la tarea 0 del jugador A, luego se ejecuta la tarea 0 del jugador B y asi sucesivamente hasta llegar a la tarea 7 del jugador B. Después de ejecutar dicha tarea se pasa a ejecutar de nuevo la tarea 0 del jugador A y asi por el hasta que no queden tareas pendientes.

Dicho mecanismo es implementado por el **Scheduler**. (Archivos **sched.c** y **sched.h**).

6.2. Contexto Inicial Tareas

Las tareas comienzan con el siguiente contexto:

- El **EIP** comienza en la primera dirección 0x16000.
- **ESP** = **EBP** = 0x27000.
- **CS** apuntando al segmento de código nivel 0.
- El resto de los selectores de segmento seleccionando el segmento de datos nivel 3.
- **CR3** = 0x27000
- **EFLAGS** = 0x202 (Interrupciones habilitadas.)

6.3. Scheduler

El scheduler cuenta con las siguientes variables que definen el estado actual del sistema:

`int jugador_actual = 0;` Indica que jugador empieza ejecutando. Nosotros por azar elegimos empezar con el jugador A.

`int tarea_actualA` Indica qué tarea se está ejecutando actualmente del jugador A.

`int tarea_actualB` Indica qué tarea se está ejecutando actualmente del jugador B.

Todas toman valores entre 0 y 7

`int posicion_zombie_A_C[CANT_ZOMBIS]` La posicion de memoria en columnas de la tarea actual del A.

`int posicion_zombie_A_F[CANT_ZOMBIS]` La posicion de memoria en filas de la tarea actual del A.

`int posicion_zombie_B_C[CANT_ZOMBIS]` La posicion de memoria en columnas de la tarea actual del B.

`int posicion_zombie_B_F[CANT_ZOMBIS]` La posicion de memoria de en filas la tarea actual del B.

`int clase_zombie_A[CANT_ZOMBIS]` La clase de cada zombie del jugador A.

`int clase_zombie_B[CANT_ZOMBIS]` La clase de cada zombie del jugador B.

`int posicion_jugador_A` La posicion en el eje Y del jugador A en el mapa.

`int posicion_jugador_B` La posicion en el eje Y del jugador B en el mapa.

`unsigned int cr3_A[CANT_ZOMBIS]` Los cr3 de cada zombie del jugador A.

`unsigned int cr3_B[CANT_ZOMBIS]` Los cr3 de cada zombie del jugador B.

```
// Inicializa la estructura del Scheduler.
int jugador_actual = 0;
int tarea_actualA = 0;
int tarea_actualB = 0;
int posicion_zombie_A_C[CANT_ZOMBIS];
int posicion_zombie_B_C[CANT_ZOMBIS];
int posicion_zombie_A_F[CANT_ZOMBIS];
int posicion_zombie_B_F[CANT_ZOMBIS];
int clase_zombie_A[CANT_ZOMBIS];
int clase_zombie_B[CANT_ZOMBIS];
int posicion_jugador_A = 1;
int posicion_jugador_B = 1;
int tipo_zombie_A = 0;
int tipo_zombie_B = 0;
unsigned int cr3_A[CANT_ZOMBIS];
unsigned int cr3_B[CANT_ZOMBIS];
unsigned int puntos_jugador_A = 0;
unsigned int puntos_jugador_B = 0;
unsigned int zombies_restantes_A = 20;
unsigned int zombies_restantes_B = 20;



unsigned short sched_proximo_indice()
{
    int i;
    if (jugador_actual == 1) //jugador A
    {
        jugador_actual = 0;
        for (i = 0; i < 8; ++i)
        {
            int pos = (i + tarea_actualA) % 8;
            if (tareas_A[pos] == 1)
            {
                    tarea_actualA = pos;
                    return (pos + 15)*8;
            }
        }

    }
    else
    {
```

```
        jugador_actual = 1;
        for (i = 0; i < 8; ++i)
        {
            int pos = (i + tarea_actualB) % 8;
            if (tareas_B[pos] == 1)
            {
                    tarea_actualB = pos;
                    return (pos + 23)*8;
            }
        }
    }

    return 8;
}
```

Código 6: Código del scheduler que determina qué tarea se debe ejecutar a continuación y prepara el entorno para ello. Si hubo un syscall fuerza el cambio a la tarea `Idle`. Además de la función que inicializa el mismo y la interfaz que exporta el scheduler para realizar el cambio de tareas. – **sched.c**

7. kernel.asm

```
    ;; Punto de entrada del kernel.
BITS 16
start:
    ; Deshabilitar interrupciones
    cli

    ; Cambiar modo de video a 80 X 50
    mov ax, 0003h
    int 10h ; set mode 03h
    xor bx, bx
    mov ax, 1112h
    int 10h ; load 8x8 font

    ; Imprimir mensaje de bienvenida
    imprimir_texto_mr iniciando_mr_msg, iniciando_mr_len, 0x07, 0, 0

    ; 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:

    ; Establecer selectores de segmentos
    xor eax, eax
    mov ax, 1010000b
    mov ds, ax
    mov es, ax
    mov gs, ax
    mov ss, ax
    mov ax, 1100000b
    mov fs, ax

    ; Establecer la base de la pila
    mov esp, 0x27000
    mov ebp, 0x27000

    ; Imprimir mensaje de bienvenida

    ; Inicializar pantalla

    call pintarTablero

    ; Manejo de interrupciones
    LIDT [IDT_DESC]

    ; Inicializar el manejador de memoria
```

```
; Inicializar el directorio de paginas
call mmu_inicializar

; Cargar directorio de paginas
mov eax, [page_dir]
mov cr3, eax

; Habilitar paginacion
mov EAX, CR0
or EAX, 0x80000000
mov CR0, EAX

; Inicializar tss
call tss_inicializar

; Inicializar tss de la tarea Idle

; Inicializar el scheduler

; Inicializar la IDT
CALL idt_inicializar

call deshabilitar_pic
call resetear_pic
call habilitar_pic
sti

; Cargar IDT

; Configurar controlador de interrupciones

; Cargar tarea inicial
mov ax, (0xD*8)
ltr ax

; Habilitar interrupciones

; Saltar a la primera tarea: Idle
mov ax, (0xE*8)
jmp (0xE*8):0
```

Código 7: Código del kernel luego de pasar a Modo Protegido. – **kernel.asm**

8. Modo debugger

Este modo sirve para saber el estado de los registros en el momento en el que se produce una excepción. El modo debug se activa presionando la tecla *zz* se desactiva de la misma manera. Si el modo debug está activado aparecerá un cartel con el estado de los registros en el momento en el que haya una excepción (no antes) y se pausa el juego hasta que el usuario presione nuevamente la tecla *z*". Si el modo debug no está activado y se produce una excepción no se debe mostrar el cartel, se desaloja la tarea que produjo la excepción y se continua con la ejecución del programa. El procedimiento cuando se ejecuta el modo debugger es el siguiente: 1) Guardar pantalla: se guarda una captura de la pantalla para que después cuando se restablezca el juego se vuelva a iniciar desde el último momento jugado. 2) Pintar debugger: Se pushean los registros necesarios y se pinta el cartel con el estado de los registros antes de que se produzca la excepción. 3) Restablecer Pantalla: Se restablece la pantalla con la última captura

hecha. Se debe guardar la pantalla por mas que este o no el modo debug activado, ya que si no está el modo debug activado, no se podria saber cual fue la última captura de pantalla antes de que se produjera la excepción.