DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

System Programming - Asimilación Cronenberg

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Alonso, Daiana	682/15	daianalonsok@gmail.com
Caballero, Tomás	628/15	tomycaballero95@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

Parte I
Resolución

1. Ejercicio 1

1.a. Definición de la GDT

La primer etapa de este trabajo consiste en completar la **Tabla de Descriptores Globales (GDT)** con cuatro entradas, que corresponden a los siguientes *descriptores de segmentos*: dos para código de nivel de privilegio 0 y 3, y dos para datos también de nivel de privilegio 0 y 3 (recordar que los niveles 0 y 3 corresponden a los niveles de privilegio de *kernel* y *usuario*, respectivamente). Los segmentos definidos nos permiten direccionar los primeros 137 MB de memoria, siguiendo un modelo de segmentación *flat*.

En este trabajo, las cuatro entradas fueron agregadas a partir del índice 8 de la GDT, ya que por especificación del problema, las primeras 7 posiciones se consideran utilizadas. Por otro lado, para cada uno de los cuatro segmentos, la *base* apunta al comienzo de la memoria (la dirección `0x00000000`) y el *límite* es la dirección `0x088FFFFF`, calculado a partir de:

$$\begin{aligned} 137 \text{ MB} &= 137 \times 1024 \text{ KB} \\ &= 137 \times 2^{10} \text{ KB} \\ &= 137 \times 2^8 \times 2^2 \text{ KB} \\ &= 137 \times 256 \times 4 \text{ KB} \\ &= 35072 \times 4 \text{ KB (35072 aumentos de a 4KB)} \end{aligned}$$

Sabiendo que $(35072)_{10} = (\texttt{0x8900})_{16}$ y que cada segmento comienza en la posición `0x00000000`, tenemos:

$$4 \text{ KB} = 4 \times 1024 \text{ bytes} = (4096)_{10} \text{ bytes} = (\texttt{0x1000})_{16} \text{ bytes}$$

$$(\texttt{0x8900})_{16} \times (\texttt{0x1000})_{16} \text{ bytes} = (\texttt{0x08900000})_{16} \text{ bytes}$$

Finalmente, restando 1 byte a este valor para poder obtener el máximo offset (en bytes), obtenemos `0x088FFFFF`. Es importante notar que, como no es posible representar este valor en el campo límite del descriptor (ya que ocupa *20 bits = 5 nibbles*), debemos poner el bit de **granularidad en 1**, y expresar el límite como incrementos de 4 KB.

1.b. Pasaje a Modo Protegido

- Luego de completar la GDT, habilitamos el pin **A20** del procesador utilizando la funciones A20_ENABLE, A20_DISABLE y A20_CHECK.
- Cargamos el registro **GDTR** con la dirección base de la GDT, utilizando la instruccion `lgdt` de ASM.
- Seteamos el bit **PE** del registro `CR0` para activar el modo protegido, e hicimos un *jump far* a la primera instrucción, utilizando el selector de segmento de código de nivel 0.
- Luego cargamos los registros de segmento `ds`, `es`, `fs` y `ss` con el selector de datos de nivel 0, y el registro de segmento `gs` con el selector que corresponde al segmento de video.
- Establecimos la **base de la pila del kernel** en la dirección `0x27000` cargando ese valor en los registros `esp` y `ebp`.

1.c. Definición del segmento de video

Definimos un segmento adicional de código para describir el área de memoria de video correspondiente a la pantalla, que podrá ser utilizada solo por el kernel y, por lo tanto, deberá tener nivel de privilegio 0. Este segmento inicia en la posición `0xBF8000` y tiene un tamaño de $80 \times 50 \times 2$ bytes $= 8000$ bytes,

mientras que el último offset posible será $8000-1 = (7999)_{10} = (\texttt{0x01F3F})_{16}$. Como podemos representar esta última dirección en 20 bits, utilizamos **granularidad 0**.
Finalmente, la GDT quedará definida de la siguiente forma:

Índice	*Base*	*Límite*	*P*	*DPL*	*S*	*D/B*	*L*	*G*	*Tipo*
8	0x00000000	0x088FF	1	0	1	1	0	1	0x08 (code, execute-only)
9	0x00000000	0x088FF	1	0	1	1	0	1	0x02 (data, read/write)
10	0x00000000	0x088FF	1	3	1	1	0	1	0x08 (code, execute-only)
11	0x00000000	0x088FF	1	3	1	1	0	1	0x02 (data, read/write)
12	0x000B8000	0X01F3F	1	0	1	1	0	0	0x02 (data, read/write)

Cuadro 1: Descriptores de la GDT

1.d. Pintar pantalla

Para pintar la pantalla se creó una función `init_screen` que se divide en 4 partes principales:

1. Pintar el borde superior negro (size: 80×1)
2. Pintar el mundo Cronemberg (size: 80×40)
3. Pintar el recuadro negro inferior (size: 80×9)
4. Pintar los rectangulos de puntaje (size: 9×3)

La función escribe en la memoria de video utilizando el selector de segmento `gs`, seteado en el archivo `kernel.asm`.

2. Ejercicio 2

2.a. Definición de la IDT

Las entradas en la **Tabla de Descriptores de Interrupciones (IDT)** serán del tipo *interrupt gate*:

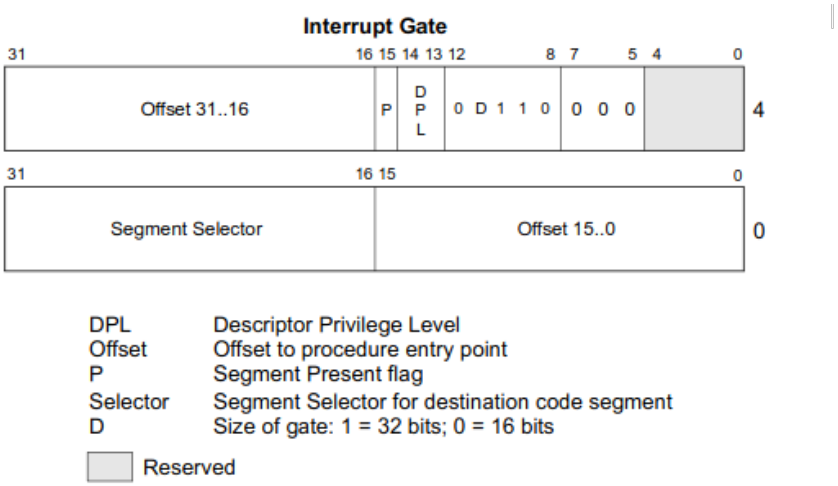


Figura 1: IDT Interrupt Gate

Definimos las entradas en la IDT para atender las 20 excepciones que posee por defecto el procesador. Para eso utilizamos la función `idt_init` donde las inicializaremos llamando a la siguiente macro provista por la cátedra, agregando como parámetro los atributos de la entrada:

```
#define IDT_ENTRY(numero, tipo_puerta)
    idt[numero].offset_0_15 = (uint16_t) ((uint32_t)(&_isr ## numero) & (uint32_t) 0xFFFF);
    idt[numero].segsel = (uint16_t) CS_RING_0;
    idt[numero].attr = (uint16_t) tipo_puerta;
    idt[numero].offset_16_31 = (uint16_t) ((uint32_t)(&_isr ## numero) >> 16 & (uint32_t) 0xFFFF);
```

En todos los casos el selector de segmento `segsel` corresponde al segmento de código de nivel 0 situado en el índice 8 de la GDT. El campo `attr` corresponde a los 16 bits menos significativos del segundo bloque, donde:

- Los bits reservados serán seteados a cero.
- Los siguientes 3 bits, como vimos en la figura, son cero.
- Como estamos trabajando en 32 bits el campo `D` lo vamos a setear en 1.
- `DPL = 0` porque serán de nivel de privilegio del kernel.
- `P = 1` para indicar que el segmento está presente.

Por lo tanto el valor del campo `attr` será $(1000\ 1110\ 0000\ 0000)_2 = (\texttt{0x8E00})_{16}$.

El offset es relativo al segmento apuntado por `segsel` que nombramos antes, y apunta a la rutina de atención de esa excepción, definida en el archivo *isr.asm*. Esta consiste en imprimir en pantalla qué tipo de error produce y detener la ejecución:

```
%macro ISR 1
global _isr%1

_isr%1:
    mov eax, %1
    push eax
    call imprimirExcepcion
    add esp,4
    jmp $

%endmacro
```

La función IMPRIMIREXCEPCION dado un número entero pasado por parámetro, imprime en pantalla el tipo de excepción correspondiente. Esto lo implementamos definiendo un arreglo de 20 elementos y en cada posición *i* pusimos el mensaje que corresponde a esa excepción.

2.b. Inicialización de la IDT

Se inicializó la IDT con la función `idt_init()` y se cargó su dirección en el registro IDTR con la instruccion `lidt`. Configuramos el controlador de interrupciones llamando a las funciones `pic_reset` y `pic_enable`. Habilitamos las interrupciones con la instrucción `sti`.Probamos el funcionamiento de la interrupción haciendo una división por cero.

3. Ejercicio 3

3.a. Interrupciones de reloj y teclado

Modificamos la macro `IDT_ENTRY` agregando un parámetro para poder indicar si las mismas tienen privilegios de usuario o de kernel:

```
#define INTR_KERNEL 0x8E00  // 1000 1110 0000 0000  D = 1, DPL = 0
#define INTR_USER   0xEE00  // 1110 1110 0000 0000  D = 1, DPL = 3

#define IDT_ENTRY(numero, tipo_puerta)
    idt[numero].offset_0_15 = (uint16_t) ((uint32_t)(&_isr ## numero) & (uint32_t) 0xFFFF);
    idt[numero].segsel = (uint16_t) CS_RING_0;
    idt[numero].attr = (uint16_t) tipo_puerta;
    idt[numero].offset_16_31 = (uint16_t) ((uint32_t)(&_isr ## numero) >> 16 & (uint32_t) 0xFFFF);
```

Agregamos en la función `idt_entry` la inicialización a las interrupciones 32 y 33 que corresponden a la interrupción de reloj y teclado con nivel de privilegio kernel y las interrupciones 137, 138 y 139 con nivel de privilegio usuario.

3.b. Rutina de reloj

Declaramos la rutina en *isr.asm* llamando a la función provista por la cátedra `nextClock` que se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla.

```
global _isr32
_isr32:
    pushad
    call pic_finish1 ; Indica que la interrupcion fue atendida
    call nextClock   ; Imprimir el reloj del sistema
    popad
    iret
```

3.c. Rutina de teclado

Escribimos la rutina asociada a la interrupción de teclado, donde llamamos a la función PRINTSCANCODE que escribimos en *screen.c* la cual si se presiona una tecla cualquiera de 0 a 9 muestra la misma en la esquina superior derecha de la pantalla.

```
global _isr33
_isr33:
    pushad
    in al, 0x60 ; Captura una tecla
    push eax
    call printScanCode ; Rutina para imprimir el scanCode
    add esp,4 ; Restaurar la pila
    call pic_finish1
    popad
    iret
```

Para imprimir en pantalla el número que corresponde en base al *scanCode* que estamos obteniendo al presionar la tecla, debemos saber cual corresponde a cada tecla del 0 al 9.

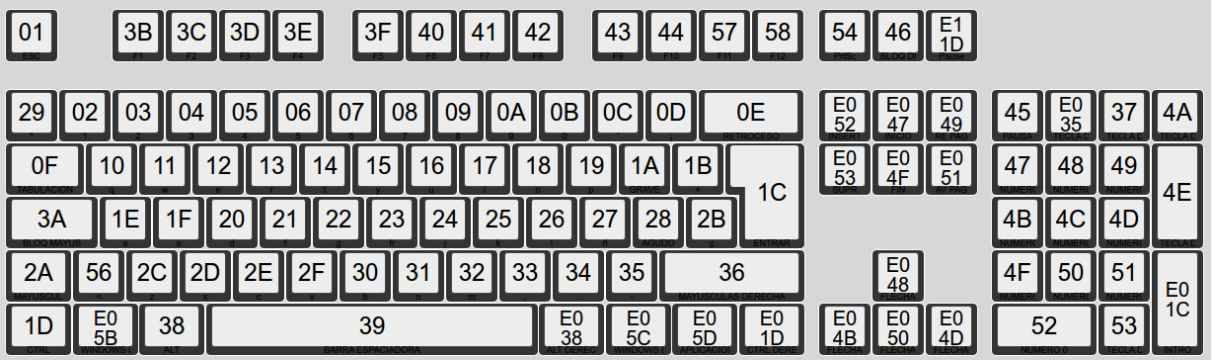


Figura 2: Distribución del teclado y su valor en hexa del *scanCode*

Como podemos ver del 1 al 9 el valor de la tecla corresponde a su *scanCode* en decimal, menos uno. El caso de la tecla '0' la setearemos por separado.

function PRINTSCANCODE(uint8_t $scanCode$)
  if hubo tecla presionada? **then**
    if $1 < scanCode < 12$ **then**
      if $scanCode = 11$ **then**
        IMPRIMIRDECIMAL$(0, 1, 80 - 1, 0)$
      else
        IMPRIMIRDECIMAL$(scanCode - 1, 1, 80 - 1, 0)$
      end if
    else
      BORRARNUMEROPANTALLA
    end if
  end if
end function

La función IMPRIMIRDECIMAL se corresponde a la función `print_dec` dada por la cátedra, que toma como parámetros un valor, su tamaño y los valores x e y, e imprime en pantalla el valor en decimal.

3.d. Agregar excepciones 137, 138 y 139

Por ahora dejamos vacías estas excepciones y solo marcamos que fueron atendidas.

4. Ejercicio 4

4.a. Identity mapping

Los primeros 4 MB de la memoria están comprendidos entre las direcciones físicas `0x00000000` y `0x003FFFFF`. Si vemos estos valores como direcciones virtuales podemos ver que el índice en el directorio de tablas siempre es 0, mientras que los de la tabla de páginas van desde el 0 hasta el 1023 (`0x3FF` en hexadecimal). Con esta información podemos ver que para mapear este área de la memoria, es necesaria una sola tabla de página cuya dirección física va a estar almacenada en el índice 0 del directorio de página del kernel.
Sabemos que el directorio de tablas del kernel comienza en la dirección `0x00027000`, mientras que la tabla de páginas lo hace en la dirección `0x00028000`.
Entonces en la rutina `mmu_initKernelDir` completamos el índice 0 del directorio de tablas con el valor `0x00028000` $\vee$ `0x000003` = `0x00028003` es decir, indicamos que la tabla de páginas tiene un nivel de privilegio 0, está presente y es de lectura/escritura (los otros atributos quedaron en 0). Luego en todas las demás entradas se setearon en 0.
Para obtener el mapeo de identidad, completamos las entradas de la tabla de páginas con las direcciones

de inicio (desde 0x0 hasta 0x3FF) con los mismos atributos que los que le asignamos a la primera entrada del directorio de páginas.

4.b. Activar paginación

Para activar paginación en kernel.asm:

- Inicializamos el manejador de memoria llamando a mmu_init, luego inicializamos el directorio de páginas llamando a la función que completamos anteriormente mmu_initKernelDir.
- Cargamos el directorio de páginas copiando en CR3 la dirección base del directorio de páginas (0x27000). Como los últimos 12 bits de esta dirección son cero, con esto también conseguimos limpiar los bits PCD y PWT del registro.
- Seteamos el bit PG (bit más significativo) de CR0 para activar paginación.

4.c. Imprimir números de libreta

Para imprimir los números de libreta en pantalla, creamos una función que llama dos veces a la función print dada por la cátedra, con el parámetro char* con el nombre y apellido de cada integrante y su número de libreta.

5. Ejercicio 5

5.a. Inicializar manejador de memoria

Declaramos una variable global *proxima_pagina_libre*. En mmu_init inicializamos esta variable con la dirección 0x00100000 que es donde comienza el área libre del kernel.
Luego completamos la función mmu_nextFreeKernelPage que devuelve el valor almacenado en *proxima_pagina_libre* y cambia su valor por los siguientes 4Kb, incrementando su valor en $1024 \times 4 = (4096)_{10} = (1000)_{16}$

5.b. Rutinas de mapeo/desmapeo de memoria

Mapear una página

function MMU_MAPPAGE(uint32_t $cr3$, uint32_t $virtual$, uint32_t phy, uint32_t $attr\_us$, uint32_t $attr\_rw$)
Obtener los índices del *Page Directory* y de la *Page Table* a partir de $virtual$
Obtener desde $cr3$ la dirección del *Page Directory*
if no existe una entrada en el *Page Directory* en ese índice **then**
Solicitar una nueva página libre del Kernel
Inicializar todas las entradas de la nueva *Page Table* en 0
Setear el bit de $presente$ en 1 en la *Page Directory*
Completar la entrada de la *Page Directory* (con permiso $Read/Write$, nivel de privilegio $attr\_us$ y con la dirección de la nueva página solicitada)
end if
Completar la entrada que corresponde de la *Page Table* con la dirección física phy, el nivel de privilegio $attr\_us$ y el permiso $attr\_rw$
Setear el bit de $presente$ en 1 de la entrada de la *Page Table*
Invalidar la caché de traducción de direcciones (TLB)
end function

Desmapear una página

Como no liberamos memoria, para desmapear una página basta con setear el bit de presente en 0.

function MMU_UNMAPPAGE(uint32_t $cr3$, uint32_t $virtual$)
Obtener los índices del *Page Directory* y de la page table a partir de $virtual$
Obtener desde $cr3$ la dirección del *Page Directory*
if existe una entrada en el *Page Directory* en ese índice **then**
Setear el bit de presente en 0 de la entrada de la *Page Table*
Invalidar la caché de traducción de direcciones (TLB)
return 1
else
return 0
end if
end function

5.c. Mapeo Mundo Cronenberg

Completamos la rutina (`mmu_initTaskDir`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, mapeando las páginas del mundo Cronenberg donde aleatoriamente estará ubicada la tarea a partir de la dirección virtual `0x08000000` y copiar en esta área el código de la tarea. Esta función devuelve la dirección del page directory de la tarea.

function MMU_INITTASKDIR(uint32_t $codigo\_original\_tarea$, uint32_t dir_fisica)
Solicitar una nueva página para el directorio de páginas
Inicializar todas sus entradas en cero
Aplicar Identity Mapping para los primeros 4MB pertenecientes al área de kernel, llamando a la función `mmu_mapPage` con permiso de supervisor, lectura/escritura y el page directory creado
Mapear el código de la tarea en su directorio, llamando a `mmu_mapPage` con la direccion virtual de la tarea, dir_fisica, permiso de supervisor, lectura/escritura y el page directory creado
Mapear con el $cr3$ actual dir_fisica con Identity Mapping para poder copiar el código de la tarea (8K)
Invalidar la caché de traducción de direcciones
Copiar el $codigo\_original\_tarea$
Desmapear para revertir el mapeo realizado
Invalidar la caché de traducción de direcciones
return dirección del page directory
end function

5.d. Rutina de prueba

Como prueba, construimos un mapa de memoria para tareas llamando a la función `mmu_initTaskDir` con una dirección fisica cualquiera dentro del mundo Cronenberg (a partir de la dirección `0x00400001`) y una dirección de código de tarea cualquiera perteneciente al area libre del kernel. Luego intercambiamos el `CR3` actual por la dirección resultado de la función (esto ya limpia los bits `PCD` y `PWT`) y llamamos a la función que imprime la libreta de los alumnos en pantalla.

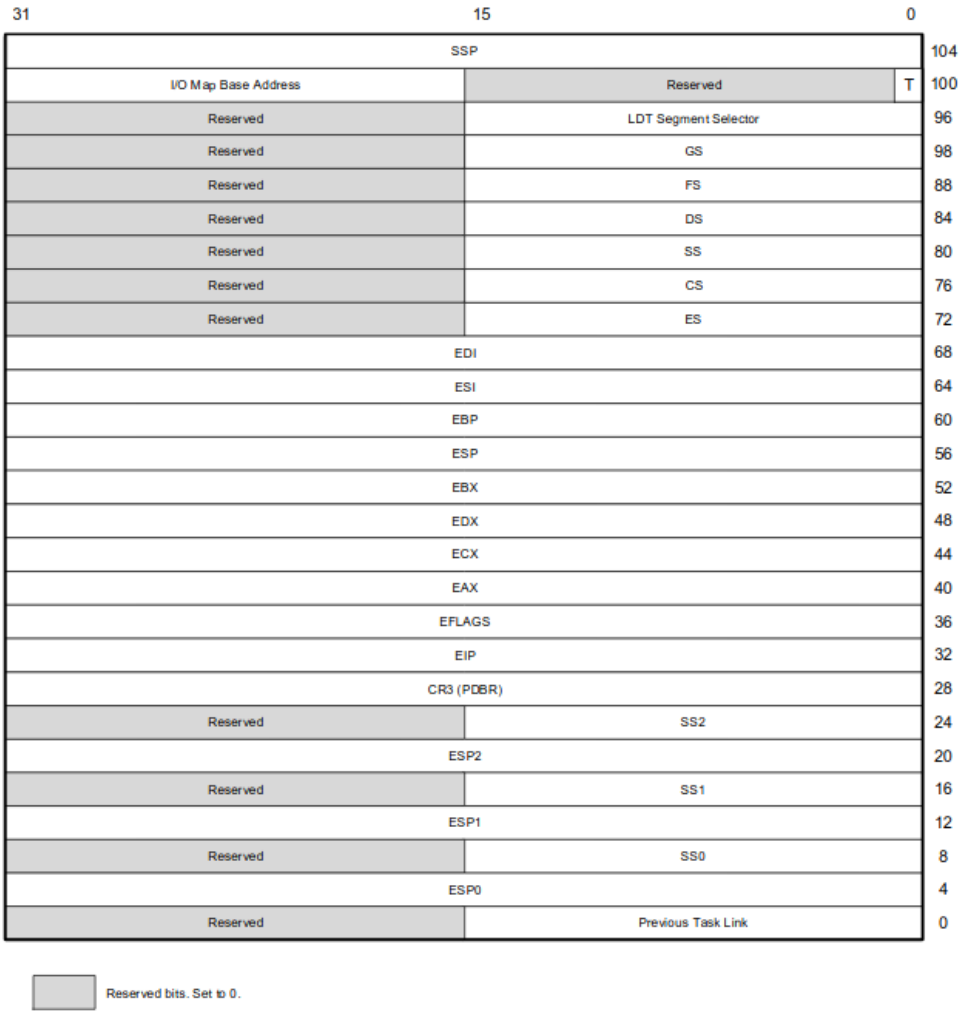
6. Ejercicio 6

Antes de poder comenzar a correr las tareas es necesario preparar algún tipo de mecanismo para poder alternar su ejecución sin perder la información asociada al estado de cada una. La forma de conseguir esto es mediante los **TSS (Task State Segment)**.

6.a. Definir entradas en la GDT

El primer paso consiste en definir entradas para los **descriptores de TSS** en la GDT: uno para la tarea inicial, que permite que se realice el primer intercambio y otro para la tarea *Idle*. Dentro de la GDT, serán asignadas en los índices 13 y 14.

6.b. Completar entradas en la TSS



Una vez que agregamos esos descriptores en la GDT, procedimos a completar la entrada de TSS de la tarea inicial y la tarea Idle en el archivo `TSS.c`.
Como cargar la tarea inicial nos sirve para proveer una TSS en donde el procesador pueda guardar el contexto actual, y tiene solo ese propósito.
Como no vamos a volver nunca al contexto de esta tarea, inicializamos en la función `tss_init()` todos sus campos en 0, y actualizamos el comienzo del TSS de esta tarea inicial en su descriptor de la GDT.
Para inicializar la tarea Idle se utilizo la funcion `tss_init_idle()` donde completamos los campos de la TSS de la siguiente forma:

- Como esta tarea debe compartir el page directory con el kernel, el `CR3` se inicializó con esa dirección.
- Como comparte la pila con el kernel, los registros `ESP` y `EBP` de esta tarea apuntan a la pila del kernel.
- El `EIP` se seteo con el valor `0x0001A000` como se indicó en el enunciado que se encuentra la dirección del código de la tarea.

- Los `EFLAGS` se setean en el valor default `0x00000202`, esto es con las interrupciones activas.
- Para el segmento de código `CS` se le asignó el selector de segmento de código de nivel 0.
- Para `DS`, `ES`, `GS`, `FS` y `SS` se le asignó el selector de segmento de datos de nivel 0
- Los registros `eax`, `ebx`, `ecx` y `edx` se inicializaron en 0.
- El campo de entrada salida `IOMAP` se seteó con el valor de puertos desactivados que es el valor `0xFFFF`.

Para completar estos campos, utilizamos la estructura dada en el archivo `TSS.h`.
Luego actualizamos el comienzo del TSS de la tarea idle en su descriptor de la GDT.

6.c. Completar la entrada en la GDT de la tarea inicial

El formato de estas entradas es el siguiente:

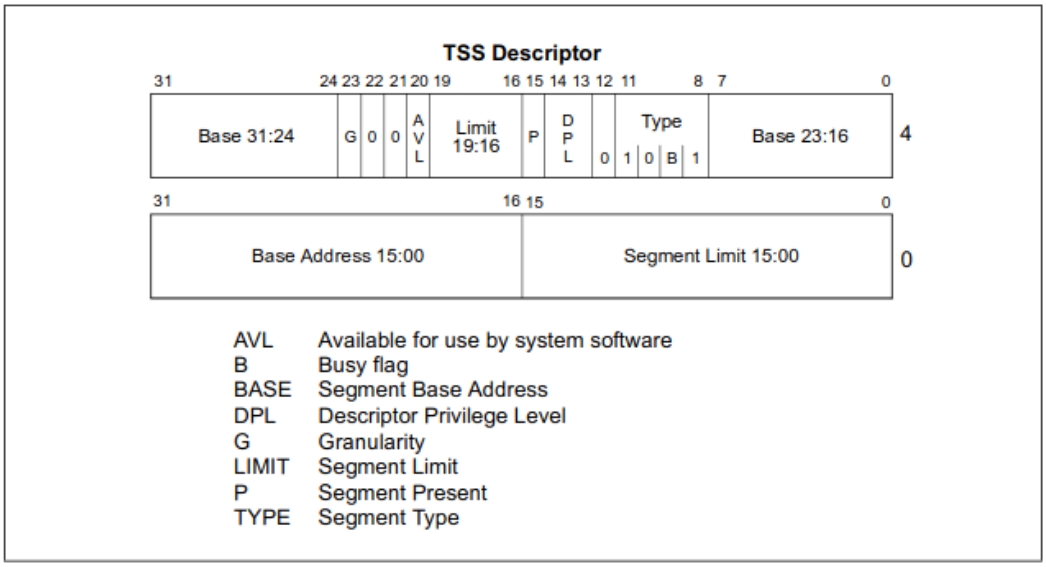


Figura 3: Descriptor de la TSS en la GDT

- El limite es `0x67` ya que el mínimo tamaño que puede tener una TSS para almacenar todo el contexto de una tarea es `0x68` (104 bits).
- El campo tipo lo completamos con el valor 9 ya que, completando el bit B (Busy) con 0, los bits quedan $(1001)_2 = (9)_{10}$.
- Se enciende el bit de presente.
- El bit `d/b` en 1 indicando que es un segmento de 32 bits disponible.
- El `DPL = 0` indicando que el nivel de privilegio será de nivel kernel, ya que saltar a una tarea es lo que hace el sistema operativo.
- Todos los demás campos no mencionados se encuentran en 0.

6.d. Completar la entrada en la GDT de la tarea Idle

Los valores de los campos en la GDT de la tarea idle serán los misma que los de la tarea inicial.

6.e. Intercambiar tareas

En `kernel.asm` inicializamos tss de la tarea inicial e Idle llamando a las funciones `tss_init` y `tss_init_idle`.Luego debemos hacer un salto intercambiando las TSS entre la tarea inicial y la tarea Idle:

- **Cargar la tarea inicial en el TR:** esto lo hacemos moviendo al registro `ax` el selector de segmento de la TSS inicial (`0x68`) y cargándola con la instrucción `ltr`.
- **Saltar a la nueva tarea (intercambio):** saltamos con la instrucción `jmp` y selector de segmento de la tarea Idle (`0x70`). Esto es el contexto de ejecución que se cargará en el procesador.

6.f. Función completar tarea

Completamos las función `tss_init_task` que inicializa una tarea cargándola en la GDT y completando la TSS con sus datos correspondientes:
Para completar la entrada en la GDT usamos la función auxiliar `tss_cargar_gdt` que define una entrada en la GDT en el índice indicado al llamar la función, actualizando la dirección base al comienzo de la TSS pasada por parámetro, y completando los demás campos con los valores vistos en el ejercicio 6.c.
Luego inicializamos el TSS de la tarea con los campos:

- `CR3 = mmu_initTaskDir` (Construimos un nuevo mapa de memoria)
- `EIP = 0x08000000` (Dirección base del código de la tarea)
- `EBP y ESP = 0x08001000` (Sumamos 0x1000 porque la pila crece desde el final de la página de la tarea)
- `EFLAGS = 0x202` (Interrupciones activas)
- `CS = 0x53` (Segmento de código nivel usuario)
- `DS, ES, FS, GS, SS = 0x5B` (Segmento de datos de nivel usuario)
- `ESP0 = mmu_nextFreeKernelPage + 0x1000`
- `SS0 = 0x48` (Segmento de datos de nivel supervisor)
- `EAX, EBX, ECX, EDX = 0`
- `IOMAP = 0xFFFF` (Puertos deshabilitados)

7. Ejercicio 7

7.a. Inicializar Scheduler

Para implementar las funciones que pongan en funcionamiento el scheduler, lo primero que se hizo fue crear una serie de estructuras que nos permitieran obtener y actualizar información sobre las tareas, los jugadores e información general del juego:

- `portal`:
  - `pos_x`: Coordenada X de la posición del portal, si el portal está inactivo su valor será -1.
  - `pos_y`: Coordenada Y de la posición del portal, si el portal está inactivo su valor será -1.
- `jugador`:
  - `portal_activo`: Portal del jugador de tipo portal.
  - `contador_portales`: Contador de cuantas veces un jugador Rick usó el arma de portales
  - `contador_puntaje`: Puntaje del jugador

- `task_info`:
  - `viva`: Indica si la tarea está viva (no fue desalojada)
  - `selector_tss`: Selector de la tarea en la GDT
  - `tipo`: Tipo de tarea (Rick y Morty de cada jugador, Cronenberg)
  - `pos_x`: Coordenada X de la posición de la tarea
  - `pos_y`: Coordenada Y de la posición de la tarea
- `info_juego`:
  - `idx_tarea_actual`: Índice en el arreglo de tareas que corresponde a la tarea actual.
  - `jugador_rojo`: Información del jugador rojo (Rick C137)
  - `jugador_azul`: Información del jugador azul (Rick D248)
  - `tareas[24]`: Arreglo de tareas del scheduler

Creamos la función `sched_init()` donde lanzamos las tareas e inicializamos las estructuras descriptas anteriormente.

7.b. Próxima tarea

Completamos la función `sched_nextTask()` que devuelve el índice en la GDT de la próxima tarea a ser ejecutada. Recorremos el arreglo de tareas `info_juego` de forma lineal, desde la posición de la tarea actual dada por `idx_tarea_actual`, buscando la siguiente tarea viva. Por lo tanto el orden de intercambio de las tareas va a estar dado por el orden de las tareas en el arreglo. En caso de que no haya tareas vivas, se ejecuta la tarea Idle.

7.c. Modificar rutinas de interrupciones 137, 138 y 139

Las tareas tienen tres formas de comunicarse con el kernel, a través de las interrupciones 137, 138 y 139.

- **Servicio "Use Portal Gun"(int 137):** Esta interrupción se resuelve llamando a la función de C `game_usePortalGun`, que dependiendo del parámetro `cross` y `withMorty`:
  - Si el servicio es llamado por un **Rick**:
    - Si `cross = 0`: Mapeamos el espacio de código del espacio apuntado por el portal a partir de la dirección virtual `0x08002000`.
      - Si `withMorty = 1`: la dirección física portal es mapeada en el esquema de paginación de su Morty correspondiente a partir de la dirección virtual `0x08002000`.
    - Si `cross = 1`:
      1. Mapeamos el espacio de código apuntado por el portal a esa posición copiando el código de la tarea Rick asociada y pisando lo que había en esa posición de la memoria.
      2. Si `withMorty = 1` el Morty asociado se desplaza de la misma forma que Rick pero desde su posición.
  - Si el servicio es llamado por un **Morty**:
    - Solo puede utilizar el servicio si su Rick correspondiente usó 10 veces su arma de portales.
    - Mapeamos el espacio de portal en ambos casos y si `cross = 1` también copiamos su código. El caso `withMorty = 1` se considera inválido y la syscall no hace nada.

  Los casos no especificados son considerados inválidos y la syscall no realiza nada.
- **Servicio "I am Rick"(int 138):** Para resolver la siguiente interrupción se utiliza la función de C `game_sumar_punto` que, en base al código del Rick que se recibe por parámetro:

1. Suma un punto al Rick correspondiente.
2. Cambia el tipo de la tarea Cronenberg asociándola al Rick que la conquistó.
3. Actualiza el puntaje en pantalla

Este servicio solo puede ser llamado por tareas Cronenberg, caso contrario no realiza nada.

- **Servicio "Where Is Morty"(int 139):** Estas interrupciones son realizadas por tareas de tipo Rick. Para resolver una interrupción de este tipo se utiliza la función de C `game_whereIsMorty` que calcula la distancia (en filas y columnas) entre el Rick que realizó la interrupción y su Morty. Los resultados son devueltos en las direcciones `&x` e `&y` recibidas por parámetro.

7.d. Intercambio de tareas

Para implementar el cambio de tareas lo que se hace es agregar la función `sched_nextTask()` a la rutina de atención del clock. De este modo, a cada tick del clock se buscara la próxima tarea activa y se procederá a atenderla. Si solo tiene una tarea activa se continua ejecutando la misma.

7.e. Modificar rutinas de interrupciones del procesador

Agregamos en la macro de interrupciones `'_isr%1'`un llamado a la función `sched_desalojar_tarea_actual()` que marca la tarea actual como desalojada, seteando en cero el atributo `viva` de la tarea.

7.f. Mecanismo de debugging

Implementamos el mecanismo de debugging para indicar por pantalla la razón de desalojo de la tarea (número de excepción) y mostrar el estado de sus registros. Este mecanismo se activa presionando la tecla 'Y' mientras las tareas estén ejecutándose.
La implementación de este modo toma como información el estado de los registros al momento de producirse una excepción. La rutina de atención de interrupciones se basa en:

- Guardar el estado de los registros utilizando la función de C `game_guardar_estado` que almacena los datos de los registros en una estructura `info_debug`.
- Indicarle al PIC que la interrupción fue atendida.
- Desalojar la tarea en cuestión utilizando la función de C `sched_desalojar_tarea_actual`, que marca la tarea como `viva = 0`.
- Ejecutar la tarea `Idle`.

Para capturar la activación/desactivación del modo debug, creamos la variable global `modo_debug_activo` para capturar este comportamiento desde la rutina de atención al teclado, que se fija si se presionó la tecla 'Y':

- Si no estaba activo (`modo_debug_activo = 0`) entonces se imprime la información utilizando la función `imprimir_pantalla_debug`.
- En caso contrario, se restaura el estado de la pantalla utilizando la función `restaurar_pantalla` y se asigna `modo_debug_activo = 0`.

Es imporante notar que, la rutina de atención de reloj fue modificada para que se fije si la pantalla de debug se está mostrando y, en ese caso, salte a la tarea Idle. En caso contrario continua normalmente con el intercambio de tareas.