UNIVERSIDAD DE BUENOS AIRES

Facultad de Ciencias Exactas y Naturales
Departamento de Computación
Organización del Computador

RECUPERATORIO AL TRABAJO PRÁCTICO NÚMERO 3

Nombre de Grupo: Napolitana con Jamón y Morrones

Alumnos:

Izcovich, Sabrina | sizcovich@gmail.com | LU 550/11

López Veluscek, Matías | milopezv@gmail.com | 926/10

Índice

1. Introducción

El siguiente trabajo práctico consiste en un conjunto de ejercicios en los que se aplican de forma gradual los conceptos de System Programming vistos a lo largo de la materia. El objetivo de éste consiste en construir un sistema capaz de correr el juego *Infección*. Para la realización del mismo, debimos crear el soporte para que los jugadores (o sea las tareas) puedan ejecutar las reglas del juego. Dado que las tareas eran capaces de hacer cualquier cosa, debimos considerar que el sistema debía tener la capacidad de capturar cualquier problema y poder quitar a la tarea del juego.
A lo largo del trabajo práctico, se utilizaron los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo enfocados sobre el sistema de protección. En lo que sigue, se detallan explícitamente los pasos realizados para lograr el objetivo mencionado.

2. Ejercicio 1

En el primer ejercicio, nos limitamos a completar la GDT conformada por un arreglo de gdt_entry tomando como ejemplo la entrada GDT_NULL provista por la cátedra. Para ello, debimos definir tres segmentos de códigos y de datos, manteniendo el segmento NULL para el buen funcionamiento de la estructura. El primero de ellos (de nivel de privilegio 0) correspondiente al kernel, el segundo (de nivel 2) al árbitro y el tercero (de nivel 3) a los jugadores. Dado que la GDT tiene un formato de segmentacion flat, la base de cada segmento comienza en 0 y el límite consiste en el tamaño de la memoria que deseamos poder acceder. Cada segmento fue completado del siguiente modo:

0xFFFF	limit[0:15]
0x0000	base[0:15]
0x00	base[23:16]
0xX	type
0x01	s
0xX	dpl
0x01	p
0x07	limit[16:19]
0x00	avl
0x00	l
0x01	db
0x01	g
0x00	base[31:24]

Cuadro 1: GDT

Dado que, como ya establecimos, se trata de una segmentación flat, las únicas diferencias entre los segmentos creados es el tipo y el DPL.
El DPL(Default Privilege Level) es el privilegio mínimo que necesita la tarea para acceder de cualquier forma a ese segmento, por lo tanto un proceso ejecutándose en el

	bit 10	bit 9	bit 8
Datos	Expand-Down	Write	Accessed
Código	Conforming	Read	Accessed

Cuadro 2: Bits de tipo

anillo de privilegio 3 no podría acceder a un segmento de privilegio 0, 1 o 2, pero cualquiera de estos otros privilegios si a uno de privilegio 3 ya que son más privilegiados. Por otro lado, el tipo del segmento es una combinación de 4 bits de los cuales el más singificativo indica si se trata de un segmento de datos(0) o uno de código(1); es importante la diferencia entre ambos porque no se puede ejecutar desde un segmento de datos así como no se puede escribir en un segmento de código, por lo que es un nivel de protección extra contra acciones maliciosas. Por otro lado los siguientes 3 bits tienen funciones que dependen del tipo de segmento:

El bit 8 es el bit que indica si fue accedido o no el segmento, por defecto se configura en 0 porque al inicio del programa no puede haber sido accedido ningún segmento.
El bit 9 es el bit que indica, en caso de tratarse de un segmento de datos, si el segmento es de escritura o lectura/escritura o, en caso de tratarse de un segmento de código, si es de ejecución o lectura. En caso de ser únicamente de ejecución, en caso de haber constantes en el código estas no podrían ser accedidas. Para que el kernel se ejecute correctamente pusimos este bit en 1 para que los segmentos de datos puedan ser escritos y los de código leídos.
El bit 10 indica comportamientos un poco más avanzados para los segmentos que no son usados en el TP y por lo tanto es configurado en 0. En el caso de ser un segmento de datos, este bit indica que el segmento "crece hacia abajo" como las pilas. En caso de ser un segmento de datos, este bit indica si se puede acceder al segmento desde un privilegio menor de manera directa o no.

Finalmente se configuraron los selectores de segmento para que CS(Code) tenga el selector de segmento que corresponde al código de nivel 0 y DS(Data), ES, FS, GS y SS(Stack) tengan el selector de segmento que corresponde al segmento de datos.

2.1. Pregunta 1:

¿Qué ocurre si se intenta escribir en la fila 26, columna 1 de la matriz de video, utilizando el segmento de la GDT que direcciona a la memoria de video? ¿Por qué?

La consecuencia de intentar escribir en la fila 26, columna 1 de la matriz de video accediendola desde el segmento declarado, es que se produce una interrupción 13(General Protection) porque se accede a memoria fuera del segmento.

2.2. Pregunta 2:

¿Qué ocurre si no se setean todos los registros de segmento al entrar en modo protegido? ¿Es necesario setearlos todos? ¿Por qué?

En el caso en el que los registros no se seteen se genera un General Protection; al menos se necesitan CS, DS y SS configurados correctamente para el funcionamiento en modo protegido, estos son el selector de código, el selector de datos y el selector de stack respectivamente. Para respetar la convención C se deben inicializar también los selectores de propósito general, ES, FS y GS, al mismo valor que DS.

3. Ejercicio 2

El ejercicio 2 nos plantea la creación de la Interrupt Descriptor Table(IDT) para responder ante las interrupciones de sistema(Interrupción 0 a 19).
Las interrupciones se producen cuando se realiza alguna acción no permitida por el sistema y por lo tanto deben proveer una respuesta, ya sea desalojando a la tarea que cometió el error o proveyendo del recurso necesario para que al retornar no se produzca el error. Puntualmente en esta instancia solo es necesario indicar de que interrupción se trata, por lo que implementamos un macro genérico que mostraba en pantalla el error y atrapaba la ejecución en un loop infinito.

Para empezar creamos las entradas en la IDT, la cual se definió en C como un arreglo de `idt_entry`, un struct con la siguiente forma:

```
typedef struct str_idt_entry_fld {
unsigned short offset_0_15;
unsigned short segsel;
unsigned short attr;
unsigned short offset_16_31;
} __attribute__((__packed__, aligned (8))) idt_entry;
```

Estas entradas estan compuestas por un selector de segmento(`segsel`) que indica el segmento de la GDT donde se almacena el handler de la tarea, un offset(partido en los primeros y los últimos 16 bits) que indica la posición dentro del segmento y un campo `attr` que contiene los atributos de la entrada. Estos atributos indican de que tipo de interrupción se trata, que permisos de usuario son necesarios para llamarla y si el segmento esta presente.
Para este kernel los atributos se configuraron para todas las entradas de interrupción de sistema(0 a 19) el flag de segmento presente en 1(Presente), DPL 0 y tipo 1110(Interrupt Gate de 32 bits). El segmento es el de código de nivel 0 y el offset es la dirección física de la interrupción ya que tenemos una segmentación flat.

Para las rutinas de atención de las interrupciones se definieron los siguientes macros en assembly:

```
%macro error 2
error%1: db "Interrupcion ",%2
error%1_len equ $ - error%1
%endmacro

%macro ISR 1
global _isr%1

_isr%1:
mov eax, %1
push ebx
mov bx, es

mov ecx, 4000
mov ax, 0x38
mov es, ax
mov ax, 0x0F00
.escribeTodo:
mov [es:ecx], ax
dec ecx
loop .escribeTodo
mov [es:ecx], ax
mov es, bx
pop ebx
imprimir_texto_mp error%1, error%1_len, 0xF, 0, 0
jmp $
%endmacro

%macro ISR_CODED 1
global _isr%1

_isr%1:
add esp, 4
mov eax, %1
push ebx
mov bx, es
```

```
mov ecx, 4000
mov ax, 0x38
mov es, ax
mov ax, 0x0F00
.escribeTodo:
mov [es:ecx], ax
dec ecx
loop .escribeTodo
mov [es:ecx], ax
mov es, bx
pop ebx
imprimir_texto_mp error%1, error%1_len, 0xF, 0, 0
jmp $
%endmacro
```

El macro `error` declara una etiqueta con una cadena de caracteres para imprimir en el mensaje de error, así como una etiqueta con el largo de la cadena, estas etiquetas son luego utilizadas por los macros `ISR` e `ISR_CODED`. Se tienen dos macros porque algunas interrupciones apilan también un Error Code que a los efectos del TP es irrelevante, luego la única diferencia entre ambos macros es que `ISR_CODED` descarta el Error Code de la tarea.
Ambos formatos de handler para las interrupciones borran la pantalla y luego imprimen el mensaje correspondiente.

Pregunta 3: ¿Cómo se puede hacer para generar una excepción sin utilizar la instrucción int? Mencionar al menos 3 formas posibles.
Para generar una excepción sin utilizar la instrucción `int` *se puede hacer uso de los siguientes métodos:*

- *Dividiendo por 0.*
- *Escribiendo a un sector de memoria que no esta mapeado en la unidad de segmentación y/o paginación.*
- *Escribiendo a un sector de la memoria para el cual no se tiene suficiente nivel de permiso.*

3.1. Pregunta 4:

¿Cuáles son los valores del stack cuando se genera una interrupción? ¿Qué significan? (Indicar para el caso de operar en nivel 3 y nivel 0).

Cuando se genera una interrupción, se debe preservar el estado actual de los

flags y todos los registros que son necesarios para retornar a la normal ejecución una vez resuelta la interrupción. Cuando se produce la interrupción puede que sucedan 2 cosas, la interrupción tiene el mismo nivel de privilegio que el proceso en el que se generó o bien tiene un privilegio diferente. Exceptuando la tarea Idle que se ejecuta en nivel 0, todas las tareas, tanto árbitro como jugadores, se ejecutan en un nivel de privilegio menor que 0, por lo que siempre que suceda una interrupción de nivel 0 se producirá un cambio de privilegio con lo que el stack de la interrupción para una interrupción de nivel 0 es del formato de cambio de privilegio. Además, las interrupciones de servicio requieren un privilegio 2 o 3 para ser utilizadas, en cuyo caso las tareas no cambiarían de nivel al pedir llamar al servicio provisto para su nivel.

Error Code
EIP
CS
EFLAGS

Cuadro 3: Stack de interrupción sin cambio de privilegio

Error Code
EIP
CS
EFLAGS
ESP
SS

Cuadro 4: Stack de interrupción con cambio de privilegio

Los datos que se almacenan por automático son entonces:

- SS: El selector de segmento donde se encuentra el stack del nivel anterior, el cual se preserva por el cambio de stack al cambiar el privilegio.
- ESP: El puntero a la posición de memoria donde estaba el ESP antes del cambio de privilegio, el cual se preserva por el cambio de stack al cambiar el privilegio.
- EFLAGS: El registro que contiene el estado de los flags del sistema. Como muchas operaciones afectan el estado de este registro, se debe conservar para asegurarse de que el estado del proceso que se interrumpió sea el mismo que al salir de la interrupción.
- CS: El selector de segmento del proceso interrumpido.
- EIP: El puntero que indica que dirección se debe ejecutar al retornar de la interrupción.
- Error Code: El código de error, en caso de existir para la interrupción, codifica información extra sobre el error para facilitar el diagnóstico del motivo de la interrupción ya que pueden haber múltiples causas.

El Error Code tiene 32 bits, de los cuales los bits 16 a 31 están reservados para mantener el stack alineado. El bit 0(EXT) indica si la interrupción es por un evento externo al procesador, el bit 1(IDT) indica si el índice, que va de los bits 3 a 15, hace referencia a un elemento de la IDT o a un descriptor en la GDT/LDT y el bit 2(TI) indica, solo en caso de que le bit 1 indique que se trata de un descriptor de GDT/LDT, en que tabla se encuentra el descriptor.

4. Ejercicio 3

Para inicializar el Page Directory y el Page Table necesarios para que el kernel se siga ejecutando una vez activada la paginación, según la configuración que se usa en este trabajo práctico, es necesario reservar dos futuras páginas de 4kb, una para el PD y otra para la PT.
Para empezar se analizaron las direcciones que se deseaban mapear como si mismas en la paginación. Estas direcciones lineales van desde la $0x00000000$ a la $0x00163FFF$, las cuales se interpretan de la siguiente forma:

10 bits	10 bits	12 bits
PDIndex	PTIndex	Offset

Cuadro 5: Interpretación de la dirección lineal por la unidad de paginación

Los 10 bits más significativos cumplen el rol de índice del PD; en todas las direcciones estos bits son 0, por lo que solo es necesario un PT. Entonces, en la página que empieza en $0x21000$ inicializamos la estructura del PD completamente vacía y luego configuramos la primera entrada para que busque la PT en la página $0x22000$ como indica la consigna. Además configuramos la entrada para que la página no sea accesible por procesos de privilegio *usuario*, sea de lectura y escritura y también se indicó como presente en memoria. Un Page Directory Entry tiene la siguiente estructura:

- Los 20 bits más significativos corresponden a los 20 bits más significativos de la dirección de la PT.
- Los siguientes 9 bits estan destinados a flags que no tienen relevancia para el TP.
- El bit 2 es el bit que indica el nivel de permiso, es decir, si la página mapeada requiere nivel de *superusuario*(0) o no(1). En relación con la unidad de segmentación, se trata de *superusuario* a los niveles 0, 1 y 2 y de *usuario* al nivel 3.
- El bit 1 indica si la página es de solo lectura(0) o si además puede ser modificada(1).
- El bit 0 indica que la página se encuentra presente en memoria(1) o no(0).

Con el PD configurado correctamente continuamos con la configuración del PT. Los índices que se usan para esta van del 0 al 355, por lo que una vez inicializada la estructura vacía, se configuraron las primeras 355 entradas para que mapeen a la misma dirección por la que se llegó.
Un Page Table Entry tiene, a los efectos del trabajo práctico, un formato idéntico al del PDE exceptuando los primeros 20 bits que son los primeros 20 bits de la dirección física que se desea mapear, siendo los 20 bits más significativos los necesarios para encontrar el inicio de cada página de 4kb.
Una vez inicializadas las entradas se activa la paginación con las siguientes lineas de código assembly:

```
mov eax, 0x00021000 ;Cargo la direccion del directorio en cr3
mov cr3, eax
mov eax, cr0
or  eax, 0x80000000 ;Activado el flag de paginacion
mov cr0, eax
```

Como se indica en el código, primero se carga en el registro CR3 la dirección del PD y luego se activa el flag de paginación en CR0, de no ser así se produciría un Page Fault al no poder encontrar el PD bien formado.

4.1. Pregunta 5:

¿Puede el directorio de páginas estar en cualquier posición arbitraria de memoria?

El directorio de páginas debe estar en una página de 4kb ya que ocupa ese espacio en memoria; dicha página debe encontrarse en el espacio de memoria accesible antes de activar la paginación pues debe ser creada antes de activarla, de no tener un directiorio de páginas al activar la paginación se produce un Page Fault ya que no tiene páginas mapeadas ni un page directory en el que buscarlas.

4.2. Pregunta 6:

¿Es posible acceder a una página de nivel de kernel desde usuario?

Una página con privilegio de superusuario no puede ser accedida por un proceso con nivel de usuario, sin embargo un proceso con nivel de super usuario si puede acceder a una página con privilegio de usuario.

4.3. Pregunta 7:

¿Se puede mapear una página física desde dos direcciones virtuales distintas, de manera tal que una esté mapeada con nivel de usuario y la otra a nivel de kernel? De ser posible, ¿Qué problemas puede traer?

Es posible, sin embargo el permitir el acceso a una misma página desde distintas direcciones virtuales con distinto privilegio podría permitir a un usuario modificar o acceder a datos que solo un superusuario debería poder acceder.

5. Ejercicio 4

En este ejercicio se inicializaron las estructuras de paginación de las tareas tanto para los jugadores como para el árbitro. Para esto, fueron utilizadas las funciones mmu_inicializar_tarea_jugador y mmu_inicializar_tarea_arbitro. Dichas funciones se encargan de inicializar el page table y page directory de cada uno. Se utilizó la misma rutina que para el kernel para crear el identity mapping pues ese sector de memoria es compartido con todos aunque con otros permisos. Además, se mapearon las páginas necesarias para la ejecución de las tareas (la página de código, pila y tablero).

- **mmu_inicializar_tarea_arbitro** mapea el código a 0x3A0000, la pila a 0x3B0000 y el tablero con permiso de escritura a 0x3C0000.
- **mmu_inicializar_tarea_jugador** invoca, a su vez, a un inicializador para cada jugador. Éstos mapean el código a 0x3A0000, la pila a 0x3B0000 y el tablero sin permiso de escritura a 0x3C0000.

Para mapear estas páginas, se utilizó la función mmu_mapear_pagina. Ésta recibe la dirección física y la dirección virtual que se quiere mapear, el CR3 y los atributos del mapeo.

```
void mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int
int *page_dir = (int *)cr3; //cargo page directory

int PDE = virtual >> 22; //busco PDE, quiero los 10 primeros bits de la dire
int *page_tab = (int *)(*((unsigned int*) &(page_dir[PDE])) & 0xFFFFF000); /

int PTE = (0x003FF000 & virtual) >> 12; //pongo en 0 virtual para obtener el
*((unsigned int*) &(page_tab[PTE])) = (0xFFFFF000 & fisica) | (0xFFF & attrs
}
```

Conociendo el CR3, se accede al page directory y, basados en los primeros 10 bits de la dirección lineal, se calcula la entrada del page directory que apunta a la tabla de páginas en donde se mapeará la dirección. De aquí se extrae la dirección del page table en el cual, usando los siguientes 10 bits más significativos de la dirección virtual, se tomará la entrada del page table a la cual mapear la dirección física. La entrada

page table, entonces, se crea con los 20 bits más significativos de la dirección física y los atributos que se recibieron por parámetro.

5.1. Pregunta 8:

¿Qué permisos pueden tener las tablas y directorios de páginas? ¿Cuáles son los permisos efectivos de una dirección de memoria según los permisos del directorio y tabla de páginas?

Las tablas y directorios de páginas pueden tener permisos de usuario o supervisor, lectura o lectura/escritura. El permiso efectivo de una dirección de memoria es, principalmente, el permiso de la tabla de páginas. Esto se debe a que los permisos del page directory indican el permiso que se tiene sobre el page table mapeado en la pde con lo cual, si se tiene permiso de lectura, se aplican los permisos de la pte. En el caso en el que el page directory indicara que el page table tiene permiso de usuario entonces el usuario no tendría acceso a las páginas mapeadas en ese table.

5.2. Pregunta 9:

¿Es posible desde dos directorios de página, referenciar a una misma tabla de páginas?

Es posible desde dos directorios de página referenciar a una misma tabla de páginas. Para que esto ocurra, debe escribirse la dirección de la tabla que se quiere referenciar en la entrada de cada directorio de tablas de páginas. Por ejemplo, esto puede ocurrir cuando dos tareas con directorios propios quieran referenciar las mismas páginas con distintos privilegios en las mismas direcciones físicas.

5.3. Pregunta 10:

¿Qué es TLB (Translation Lookaside Buffer) y para qué sirve?

Consiste en una tabla en el proceso de memoria que contiene información sobre las páginas en memoria que el procesador accedió recientemente. La tabla hace remisión a direcciones virtuales del programa con las direcciones correspondientes en la memoria física que el programa usó más recientemente. El TLB permite mayor velocidad computacional debido a que facilita el proceso de direcciones que toman lugar independientemente del canal normal de traducción de direcciones.

6. Ejercicio 5

En este ejercicio, se implementaron las rutinas de atención de interrupción de el reloj, el teclado, int 0x80 e int 0x90. Para esto, hubo que reconfigurar el pic para evitar los conflictos de interrupciones externas al procesador. Ésto se realizó con deshabilitar_pic, resetear_pic y habilitar_pic, las cuales fueron provistas por la cátedra. Una

vez reconfigurado el pic, las interrupciones de reloj y teclado llegan por las interrupciones 0x32 y 0x33 de la idt respectivamente.
Por otro lado, los handlers de las interrupciones debieron ser alterados de manera tal que, al generarse la interrupción se invoque a la función del scheduler 'sched_remover_tarea' para que desaloje la tarea de la cola y detener su ejecución. La tarea de reloj invoca a la función proximo_reloj que muestra en pantalla el pasaje de los ticks. La rutina de teclado se encuentra vacía pues no hay tareas ejecutándose que se puedan detener con esta interrupción.
Las interrupciones de servicio (int 0x80 e int 0x90) reciben un set de parámetros a través de los registros EAX, EBX, ECX, EDX y ESI. En el caso de la interrupción 0x80, el jugador puede solicitar uno de dos servicios: duplicar y migrar los cuales se encuentran provistos por la cátedra. Para esto, se apilan, según la convención C, los parámetros recibidos para otorgárselos a la función del servicio. Para poder cumplir con esto, hubo que averiguar el proceso actual por medio de la función get_actual del Scheduler, ya que es uno de los parámetros requeridos por las funciones.
En el caso de la interrupción 0x90, las funciones iniciar y terminar no requieren de ningún parámetro por lo que se invocan sin apilar nada. Estas funciones se explican en detalle más adelante.

6.1. Pregunta 11:

¿Qué pasa si en la interrupción de teclado no se lee la tecla presionada?

En el caso en el que la interrupción del teclado no lea la tecla presionada, dicha interrupción tendría un único comportamiento dado que no existe manera de saber qué tecla fue pulsada.

6.2. Pregunta 12:

¿Qué pasa si no se resetea el PIC?

En el caso en el que el PIC no sea reseteado, más de una interrupción puede ser atendida por el mismo handler. Esto se debe a que el PIC se encuentra mapeado por defecto sobre interrupciones del sistema, por lo tanto, hay que configurarlo para que quede fuera del rango de las interrupciones de Intel.

7. Ejercicio 6

En este ejercicio se pide la inicialización de las diferentes estructuras para el intercambio de tareas y el comienzo de la ejecución del Scheduler. El Scheduler es un componente del sistema operativo que administra el inicio de la ejecución de las

tareas, órden de ejecución de las tareas y detenerlas en caso de que se haya producido un error durante su ejecución. Para que este funcione se debe poner en marcha el intercambio de tareas. Es necesario entonces la creación de las TSS correspondientes a las tareas Idle, Árbitro y Jugadores y una para la "tarea inicial" inexistente, la cual, al no haber tareas ejecutándose al inicio y tener que guardarse obligatoriamente el estado actual, cumple la única función de permitir el inicio de ejecución de la tarea Idle.

Un Task-State Segment(TSS) es un segmento de 104 bytes que almacena el estado de todos los registros relevantes a la ejecución de una tarea con el siguiente formato:

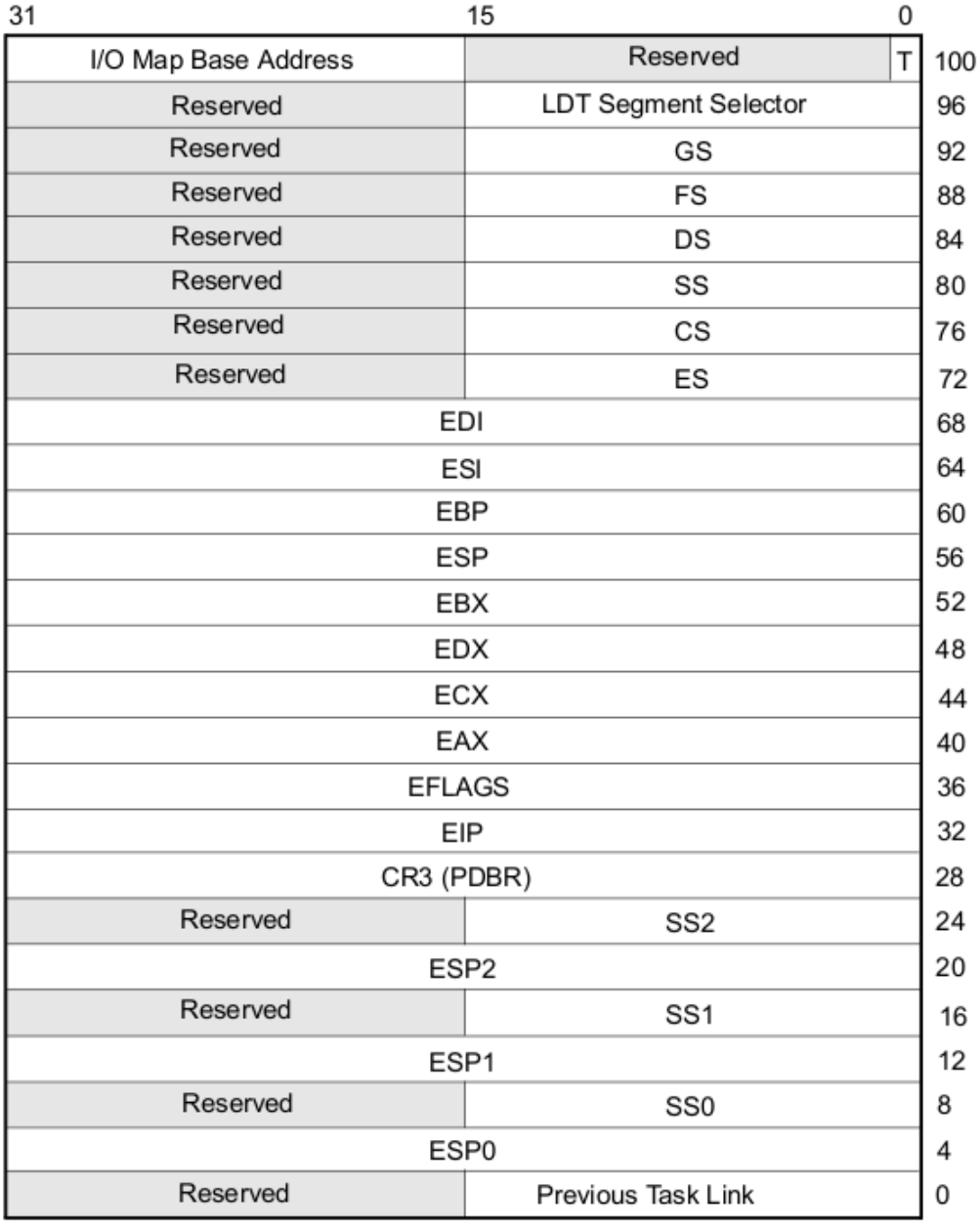

Figura 1: Formato del TSS

Los bits reservados deben ir en 0.

Para realizar el cambio a una tarea, se debe inicializar la estructura correctamente, en otras palabras configurar los descriptores de segmento correctamente para que carguen los segmentos adecuados al permiso de la tarea y el SS0 al segmento de Datos de nivel 0, configurar el CR3 para que cargue el correcto Page Directory, los registros ESP y EBP al comienzo de la pila de la tarea, ESP0 al comienzo de la pila de nivel 0 para cuando se cambie de nivel por una interrupción y finalmente el EIP para que la tarea inicie su ejecución desde el inicio de la tarea. Además se configura EFLAGS en $0x202$ para habilitar las interrupciones y el valor de IOMAP en 0xFFFF.

Cada tarea(Idle, Jugadores y Árbitro) tienen una de estas estructuras inicializadas. Además existe un TSS que esta sin inicializar para la tarea inicial, la cual, como se dijo antes, solo sirve para empezar el intercambio de tareas; esta tarea, sin embargo, tiene un descriptor de TSS en la GDT al igual que las tareas "reales" pues se necesita de ese descriptor para que el procesador encuentre la posición de memoria donde debe guardar el estado. Por lo tanto se crean 7 nuevas entradas en la GDT, cada una con un descriptor de TSS el cual se diferencia de los otros descriptores de segmentos utilizados por ser un segmento de sistema y no de datos/código. Todos los segmentos tienen un tamaño de 104 bytes y tienen como base la posicón de la estructura declarada en el archivo tss.h y tss.c.
Antes de hacer el salto a la tarea Idle se modificó `mmu_inicializar_kernel` para que mapeara la página del código de Idle, ya que este comparte su Page Directory con el Kernel. Una vez mapeada esta página se modificó kernel.asm para realizar el salto a la tarea Idle:

```
mov ax, 0x40
ltr ax

jmp 0x48:0
```

El registro TR(Task Register) almacena el selector de segmento correspondiente al TSS de la tarea que se esta ejecutando. Como no hay ninguna tarea aún, lo configuramos para que apunte al TSS de la "tarea inicial" y luego ejecutamos un far jump a la tarea idle, proporcionando el selector de segmento de su TSS.

7.1. Pregunta 13:

Colocando un breakpoint luego de cargar una tarea, ¿Cómo se puede verificar, utilizando el debugger de Bochs, que la tarea se cargó correctamente? ¿Cómo se llega a esta conclusión?

Para verificar que una tarea fue cargada correctamente, se puede pedir el $info\ tss$ en BOCHS y verificar que en el TR se encuentra el selector de segmento que apunta al

descriptor de tss de la tarea en la GDT. Por otro lado, el bit de Busy debe hallarse encendido.

7.2. Pregunta 14:

¿Cómo se puede verificar que la conmutación de tarea fue exitosa?

Dicha verificación puede realizarse a partir de los registros TR y CR3. Si se deseara comprobar que la tarea desde la que se saltó guardó correctamente los valores en la tss, se debe volver a saltar a ella misma y comprobar que dichos valores sean los mismos.

7.3. Pregunta 15:

Se sabe que las tareas llaman a la interrupción `0x80` y por `0x90`, ¿qué ocurre si esta no está implementada? ¿Por qué?

En el caso en el que se llame a la interrupción 0x80 y 0x90 y éstas no se encuentren definidas, se genera un General Protection.

8. Ejercicio 7