Trabajo Práctico 3

Martes 15 de Julio de 2014 — Organización del computador II

System Programming - TronTank

Grupo: Grecia/Gyros

Integrante	LU	Correo electrónico
Guido Rajngewerc	379/12	guido.raj@hotmail.com
Martín Caravario	470/12	martin.caravario@gmail.com
Federico Hosen	825/12	fhosen@hotmail.com

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

Índice

Resumen

Para realizar este informe, decidimos abarcar cada uno de los ejercicios, y explicar punto por punto las decisiones que tomamos y las funciones que creamos para su resolucion. Finalmente decidimos aclarar algunas observaciones y algunos problemas con los que nos encontramos a medida que ibamos resolviendo los ejercicios planteados. Muchos de los ejercicios no fueron realizados en el orden pedido, ya que en varios casos nos adelantamos a lo que nos pedian y creabamos funciones que resolvian mas de un ejercicio.

1. Ejercicio 1

A) Este ejercicio no presentó mayor dificultad, simplemente modificamos el archivo **gdt.c** agregando las nuevas entradas a la gdt. La idea es que el sitema utilize flat segmentation como indica el enunciado del trabajo. Por esto es necesario que cada segmento describa 733 MB, por lo cual debimos ajustar el limite de todos los segmentos definiendolo en 0x2DCFF. Con respecto al archivo **gdt.h** solamente modificamos la definición de la GDT COUNT, ajustandola a la cantidad de entradas que agregamos a la gdt.

B) Para este ejercicio comenzamos habilitando la linea de control A20 con el objetivo de poder direccionar direcciones sobre el MB de memoria. Esto lo hicimos con la función proporcionada por la catedra, habilitar-a20.
Luego debimos cargar la gdt creada anteriormente con la instruccion de intel lgdt, a la cual le pasamos la direccion donde se encontraba nuestro descriptor de la gdt.
Lo siguiente que hicimos fue habilitar la segmentación, para lo cual desarrollamos el siguiente código:

```
mov eax,cr0
or eax,1
mov cr0,eax
```

Al hacer or con eax, lo que estamos haciendo es poner en 1 el bit de segmentación, para habilitarla y dejando los otros bits como estaban originalmente. Notese que solo se puede trabajar sobre cr0 desde el registro eax.
Despues de esto definimos una etiqueta dentro del código a la cual saltaremos con un far jump para pasar al modo protegido. Esto lo hicimos con el siguiente código:

```
jmp (9*0x8):inicio_modo_protegido
```

Debemos aclarar que el cálculo que realizamos para la base se debe a que estamos accediendo a la novena entrada de la gdt (la cual contiene el código de nivel de prioridad 0), y todas las entradas tienen un tamaño de 8 bytes. El offset se corresponde a la etiqueta creada que se encuentra inmediatamente debajo de esta instrucción. Al pasar al modo protegido también debimos indicar que empezabamos a trabajar con 32 bits en vez de 16, como usabamos en modo real. Finalmente seteamos la pila del kernel en la posición 0x27000, como se pedia en el enunciado. Esto lo hicimos moviendo tanto el stack pointer como el base pointer a esa dirección, ya que la pila esta vacía.

C) La nueva entrada de la gdt creada para este ejercicio posee los mismos atributos que la anterior entrada definida de datos de nivel 0. Los únicos cambios que realizamos fueron el de la base y el del limite, ajustandolo a los representado en el siguiente gráfico

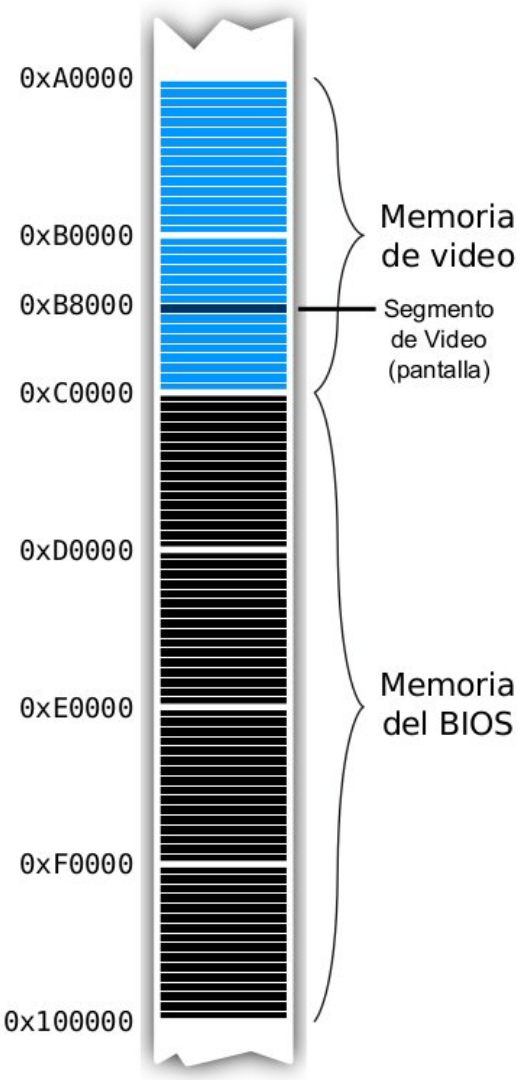


Gráfico de la memoria

Notese que a la hora de definir limites realizamos la cuenta y tomando la idea de que estamos frente a una segmentacion *flat* la misma cuenta nos sirvió para todas las entradas. Obviamente con excepción de la ultima entrada que ya mencionamos.

D) Para este ejercicio creamos una funcion, llamada *limpiar_mem_pantalla*, la cual se encarga de limpiar la memoria. Luego realiza un llamado a las limpiar_pantalla que seteará todos los caracteres en 0b00000000 que corresponde al caracter ASCII del espacio. Esta misma función pintará la pantalla de verde y utilizará funciones auxiliares que analogamente irán pintando los sectores de la pantalla como a nosotros nos parezca conveniente.
En todos los casos las cuentas de offsets y demás las realizamos a mano teniendo en cuenta que cada lugar de la pantalla es representado por 2 bytes.

2. Ejercicio 2

A) Para este punto nos basamos fuertemente en el uso de macros. En primer lugar usamos el macro sugerido para la creación de las entradas de la IDT que íbamos a necesitar. Para esto rellenamos los campos correspondientes asignando como selector, al segmento de la gdt correspondiente a código de nivel 0, y en los atributos DPL 0 y bit de presente encendido. Con respecto al offset, completamos con la dirección donde comienza el handler de la interrupción correspondiente.
La función inicializar se encarga de inicializar cada entrada de la idt con sus correspondientes atributos y handlers, los cuales fueron configurados para mostrar por pantalla el número de interrupción. Para esto modificamos el archivo isr.asm al cual le agregamos a cada handler un llamado a la función que imprime por pantalla aportada por la cátedra.

B) Para probar las interrupciones, lo que hicimos fue llamar desde assembler a alguna interrupción que hayamos definido en nuestra IDT, como por ejemplo la 15, y observamos cómo se imprimia por pantalla el número de interrupción.

3. Ejercicio 3

A) Para este punto lo que hicimos fué crear 3 funciones las cuales se encargaron de pintar cada sección del mapa, con los colores correspondientes a los pedidos en el enunciado. La idea fué modularizar el ejercicio, utilizando una función que pinte cada zona por separado. La función *pintar_gris_grande* se encargó de pintar de gris el rectángulo (8,51) , (8,79) , (38,51) , (38,79), mientras que con *pintar_gris_chico* pintamos de gris el rectángulo (47,53), (47,69), (49,53), (49,69). Finalmente nos encargamos del último rectángulo con *pintar_rojo_chico* pintando de rojo el rectángulo (7,51), (7,79). Si se lee el código se notará que antes de comenzar a trabajar en cada sector debimos calcular a mano las variables que ibamos a necesitar. Las mismas incluían la cantidad de iteraciones que iban a ser necesarias y los offsets para los saltos a la siguiente linea.
Luego de pintar los sectores correspondientes, creamos la función *escribir_pantalla*, la cual anida una serie de llamados al macro de impresión proporcionado por la cátedra lo que nos permite mostrar en pantalla los nombres de los registros y las variables que luego imprimiremos.

B) En este ejercicio lo que hicimos fué realizar la funcion *inicializar_dir_kernel*. Esta función lo que hace es llamar a map_identity con el parámetro global dir_kernel que contiene la direccion 0x27000 donde comienza el directorio de páginas del kernel, tal como lo pide el enunciado.
Map_identity se encargará de crear 4 tablas de segundo nivel, para poder mapear las direcciones correspondientes, es decir de 0x00000000 a 0x00DC3FFF. Estas tablas las pedimos de la memoria física con la funcion *get_free_page* que devuelve la primer posición libre de memoria (tamaño 4K), del sector "Área Libre"(0x100000 - 0x3FFFFF). Tambien crea 4 entradas en el directorio de páginas para estas tablas inicializandolas con el bit de presente en 1, atributos de lectura/escritura y permisos de usuario. En la dirección pusimos la base de la tabla de segundo nivel correspondiente, shifteada 12 bits a la derecha por ser múltiplo de 4kb.
Después de encargarse de las entradas del directorio de páginas la función inicializa todas las entradas de las tablas de segundo nivel con el bit de presente en 1, los atributos de lectura/escritura y permisos de supervisor. Como hay entradas de la cuarta tabla que no son utilizadas (a partir de la entrada 452), lo que hace la función es inicializarlas con el bit de presente en 0 para no tener problemas con las estructuras. Finalmente completamos el campo page_frame de cada entrada de segundo nivel con la variable destino (la cual fue inicializada en cero), que aumentamos en 1 por cada ciclo del for con el objetivo real de incrementarlas 4kb ya que, como ya dijimos, este campo de la dirección esta shifteado 12 bits a la derecha por ser múltiplo de 4kb, garantizando de esta manera que queden mapeadas con identity mapping las direcciones virtuales a las físicas.

C) Para activar paginación, lo que hicimos fue cargar en el cr3, la base del directorio de páginas del kernel, y activar el último bit del registro cr0, es decir, el bit correspondiente a paginación. Para esto utilizamos el siguiente codigo:

```
Cargamos directorio de páginas
mov eax, 0x27000
mov cr3, eax

Habilitamos paginación
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Cabe aclarar que el OR utilizado se encarga de setear en 1 el bit mas significativo del cr0 y deja como estaban los bits restantes. También destacamos que los primeros 12 bits del registro cr3 van en cero ya que son ignorados.

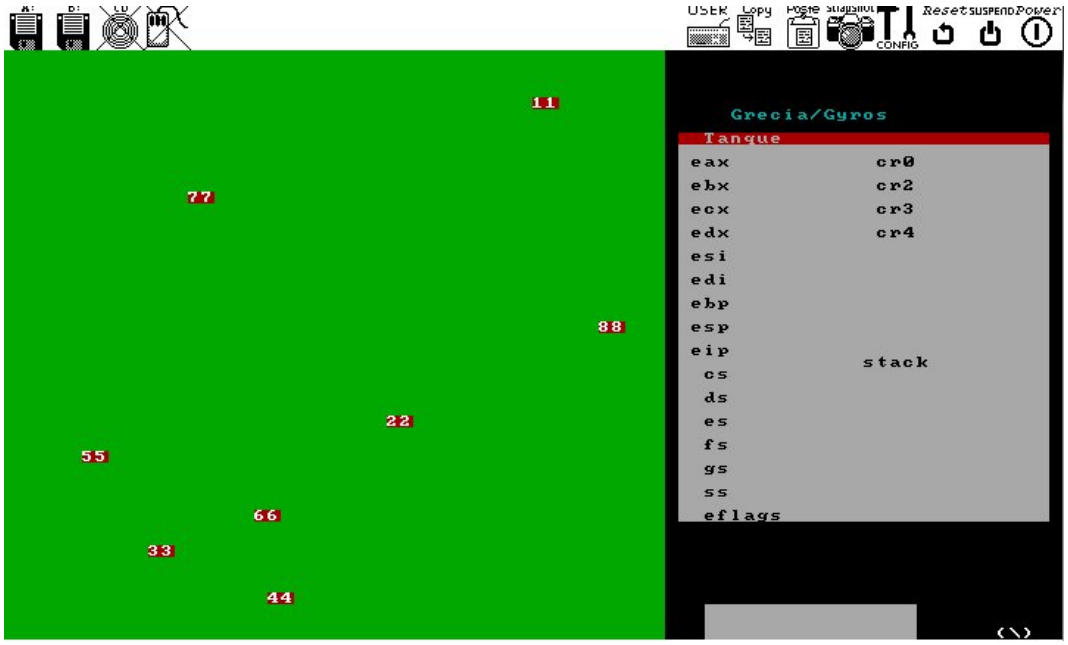
D) Para este ejercicio lo que hicimos fue agregar en la funcion *escribir_pantalla* el código necesario para poder imprimir por pantalla el nombre de nuestro grupo. Tomamos la decisión de alinearlo en la parte superior de la sección donde se muestran el estado de los registros.

4. Ejercicio 4

A) y B) Para estos ejercicios creamos la función *mmu_inicializar*, la cual se encarga de 3 cosas: inicializar el directorio del kernel, asignar páginas libres para los directorios de las tareas e inicializarlas.
Para inicializar el directorio del kernel, llamamos a la función realizada en el ejercicio anterior que nombramos *inicializar_dir_kernel*. El paso siguiente fue crear la función *inicializar_directorios*, la cual se encarga de pedir 8 páginas del kernel para los directorios de las tareas, utilizando la función *get_free_page* que ya detallamos en la explicación del ejercicio anterior.
Para definir las direcciones de memoria física donde cargar los códigos basicamente los *hardcodeamos* de manera que al representar esos espacios de memoria en el mapa estos no describan ningun tipo de comportamiento previsible. Es decir que su aparición en el mapa no pueda adivinarse a menos que se conozca el código de antemano (de todas formas el

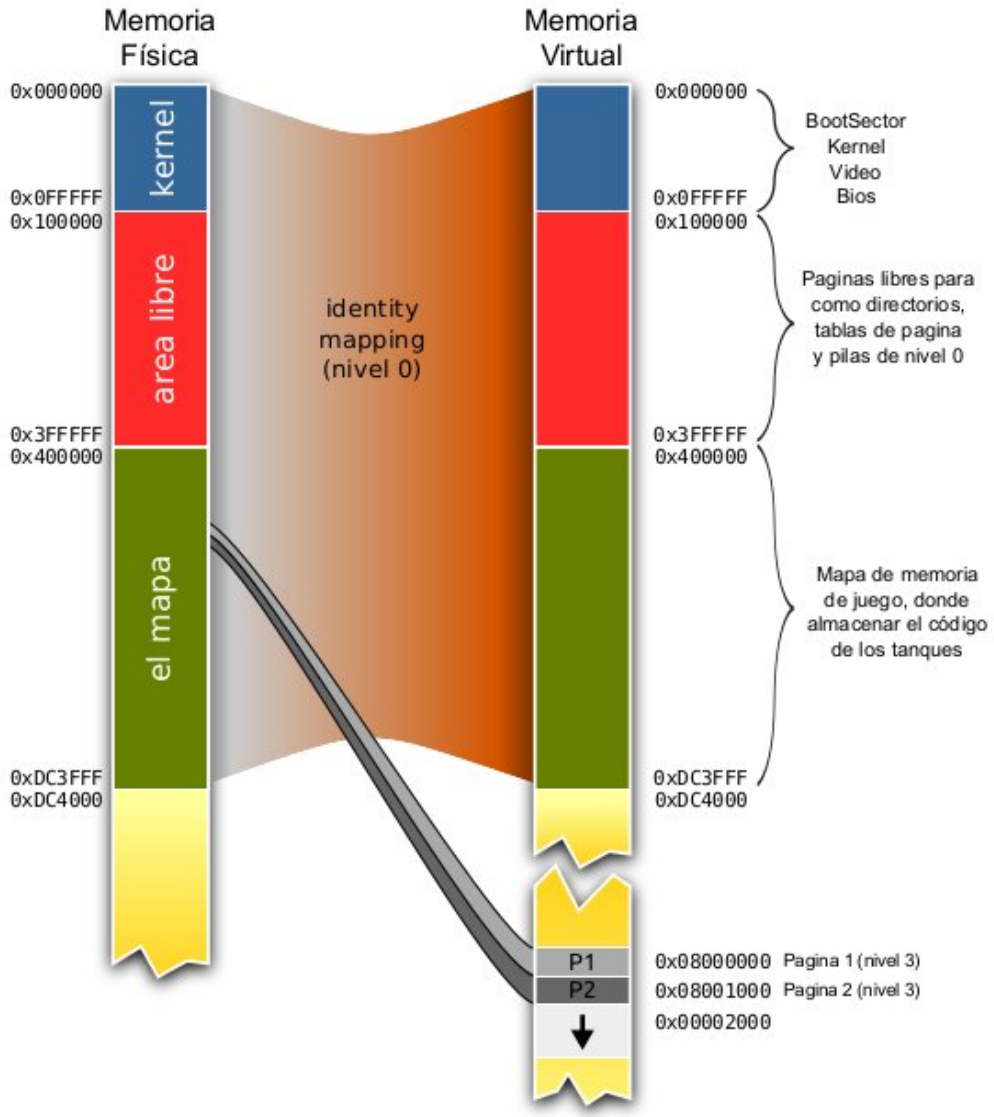
comportamiento es sincrónico, siempre que se corra el código aparecerán en el mismo lugar). Las direcciones que utilizamos son:

$$\text{dir\_fis\_1} = 0x4F0000;$$
$$\text{dir\_fis\_2} = 0xA2B000;$$
$$\text{dir\_fis\_3} = 0xC3F000;$$
$$\text{dir\_fis\_4} = 0xD10000;$$
$$\text{dir\_fis\_5} = 0xAAA000;$$
$$\text{dir\_fis\_6} = 0xBB1000;$$
$$\text{dir\_fis\_7} = 0x666000;$$
$$\text{dir\_fis\_8} = 0x8AB000;$$



Y así es como se ven dichas direcciones en el mapa

Habiendo conseguido las páginas para los directorios de las tareas, lo que hicimos fue un ciclo en el cual inicializabamos estas tareas, tal como lo pedía el mapa de memoria de cada una. Esto lo resolvimos utilizando una función llamada *mmu_inicializar_dir_tarea* que se encarga de hacer identity mapping sobre el kernel, mediante la funcion map_identity, previamente detallada. Después de haber hecho el identity mapping, copiamos el código de la tarea a la zona correspondiente del mapa, mediante un ciclo en el cual precalculamos la cantidad de iteraciones, ya que sabíamos que teniamos que mapear dos páginas por tarea. Finalmente mapeamos las direcciones virtuales en donde se encuentra el código de cada tarea (dirección 0x08000000), con su respectiva dirección física. Esto lo hicimos mediante la función *mmu_mapear_pagina* que se encarga de mapear, con privilegios de usuario, la dirección virtual a la dirección física, utilizando el cr3, todos pasados como parámetros.



Imágen de como deja la función mmu_inicializar_dir_tarea al mapa de la tarea pasada como parámetro

C) La funcion mmu_mapear_pagina lo que hace es descomponer la dirección virtual pasada como parámetro y luego entrar a la posición del directorio indicada por dir_i. Teniendo la entrada del page directory, lo que hacemos es chequear si tiene el bit de presente en 1, para ver si existe alguna tabla asociada a esa entrada. Si no la tiene lo que hacemos es llamar la función map_new_table la cual crea una tabla de segundo nivel inicializando todas sus entradas en cero y la relaciona con la entrada del directorio pasado como parametro. Luego de esto nos posicionamos sobre la entrada de la page table indicada en table_i e inicializamos sus campos con la información correspondiente, y la dirección física shifteada a derecha 12 bits.
Para la funcion mmu_unmapear_pagina lo que hicimos fue repetir del código de mapear_pagina el proceso hasta conseguir la entrada de la tabla de segundo nivel. Una vez posicionados sobre la entrada, cambiamos a cero el campo de presente para desmapearla.
Cabe aclarar que para la realización de esta función asumimos como precondición que la dirección virtual pasada como parámetro estaba realmente mapeada a una dirección física, ya que si no estaríamos escribiendo sobre cualquier lugar de la memoria. También utilizamos la función tlbflush (proporcionada por la catedra) al final de cada función, tal como fue indicado en la nota del ejercicio, ya que modificamos el directorio y tabla de páginas.

D) Este ejercicio no se ve representado en la entrega final, pues una vez comprobado el correcto funcionamiento del mismo reutilizamos el código en varias de las funciones que describimos en los puntos anteriores.

5. Ejercicio 5

A) Para este ejercicio agregamos a la IDT las entradas 32,33 y 52 correspondientes al reloj, teclado, y syscall respectivamente. Estas entradas las inicializamos con el macro previamente mencionado en el ejercicio 2, ya que todas

tienen DPL 0 y su handler en la posición de memoria correspondiente. Lo único que modificamos fué el DPL de la interrupcion 52, ya que por defecto el macro la inicializa con cero, es por esto que lo cambiamos a 3, logrando así que las tareas puedan llamarla. Esto lo hicimos con el siguiente codigo:

idt[0x52].attr = 0xEE00;

B) Este punto en particular no presento mayor complejidad dado que todo lo que necesitabamos era realizar un *call* a la función provista teniendo en cuenta siempre que por ser una interrupción de hardware necesitaríamos liberar el pic con el uso de la otra función dada, *fin_intr_pic1*.

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

C) No nos parece que haga falta ahondar mucho en este punto. La caracteristica era simple y la implementación que realizamos la reutilizamos para mostrar el número de tanque que se quiere ver en la interrupción del teclado.

D) Nuevamente este ejercicio no era particularmente complicado. El código era simple y quedó inutilizado al implementar las syscalls. No implicaba mucho más que el código que detallamos en el punto B solo que cambiando el llamado a *proximo_reloj* por *mov eax, 0x42*

6. Ejercicio 6

A) D) E) F): Para estos ejercicio lo único que hicimos fue agregar 3 entradas nuevas a la gdt 3, todas con los mismos atributos y privilegios. En el campo tipo, completamos con 9, ya que el bit de busy va en cero, mientras que en granularity pusimos 0, ya que el límite fue expresado en bytes como 0x67. El límite en hexa representa el número 103 en decimal, ya que una tss ocupa 104 bytes, pero a eso hay que restarle 1, pues el primer offset válido es el cero. Finalmente completamos el campo presente con 1, y el DPL con 0 para que solo pueda se acceder con EPL 0.

B): Para este ejercicio completamos la estructura tss_idle con los valores correspondientes del enunciado. Esto lo hicimos dentro de la función tss_inicializar. Como esta tarea debe correr en nivel cero, le asignamos los respectivos selectores de datos y de código con nivel cero. Después, completamos los registros de la idle, inicializandolos todos en cero. También inicializamos la pila de nivel 3 de la tarea, con esp y el ebp de la pila del kernel (aunque creemos que la idle nunca necesitaría datos de nivel 3). Luego le asignamos al selector de segmento de pila de nivel 0 el selector de datos de nivel 0. Finalmente utilizamos la función rcr3(),proporcionada por la cátedra, con el objetivo de cargarle a la idle el cr3 del kernel, tal como lo pide el enunciado. Como pila de nivel 0, se le asignó la pila del kernel (0x27000) Al cargar el cr3 del kernel nos aseguramos que la tarea esta en identity mapping con el kernel, ya que comparten el mismo directorio de páginas, o sea, el mismo mapa de memoria. Con respecto al eip, lo inicializamos con el valor 0x20000, ya que es ahí donde comienza la tarea.

C) Lo que hicimos en este ejercicio fue agregar dentro de la función inicializar_tss, un ciclo para inicializar las 8 tareas en las direcciones correspondientes al enunciado. A diferencia de la idle, estas tareas corren en nivel 3, es por eso que asignamos los segmentos correspondientes a código y datos de nivel 3. Luego inicializamos sus registros con el valor cero y el eip con la dirección de inicio del código, a la cual le fué asignada la dirección virtual 0x08000000, donde comienza el código de las tareas. Para la pila de las tareas, le asignamos la dirección donde comienza el código + 8kb, pues la tarea ocupa 2 páginas y la base de la pila debe estar ubicada en la base de la tarea. Con respecto a la pila de nivel cero, pedimos una página con la función get_free_page, y a esa página le sumamos 4kb, para que esp0 se ubique en la base de esa página pedida, como pide el enunciado.
La función inicializar_dir_usuario se encarga de cargar en el cr3 de la tss de cada tarea el valor que ya fue inicializado en mmu_inicializar. No hace falta inicializar el mapa de memoria pues ya fue creado anteriormente anteriormente, cuando inicializamos todo lo respectivo a memoria, específicamente en *inicializar_dir_tarea*.

G) Desde el kernel, una vez que tenemos todas las estructuras cargadas y listas, sólo queda cargar el task register de la tarea inicial para luego saltar a la tarea idle y así dar comienzo al juego. Utilizamos la instrucción *ltr* y el parámetro correspondiente a la entrada de la GDT donde cargamos la TSS de la tarea inicial.
Acto seguido debimos realizar el salto a la tarea idle. Para esto ejecutamos un *jump far* al selector de la GDT donde previamente habíamos cargado el contexto inicial para que la idle pudiera ejecutarse sin problemas.

7. Ejercicio 7

Observación 1: el funcionamiento detallado del scheduler se describe en la sección 8.1

Observación 2: las tareas son numeradas del 0 al 7, pero son mostradas de 1 a 8.

A) Debido a la implementación que decidimos realizar era necesario crear una serie de estructuras. Estas fueron:

- Tarea_actual: corresponde al número de la tarea cargada en la TSS que está ejecutandose. Es decir, es la tarea corriendo actualmente.
- Tarea_anterior: es la tarea que cuyo contexto está actualmente cargado en la TSS que no está en uso. Su contexto no ha sido guardado aún en el arreglo de TSS.
- Tareas_tss (tss.c): Es el arreglo donde guardaremos las TSS de las tareas.
- Cant_validas: La variable cant_validas representa la cantidad de tareas de usuario que no han sido desalojadas (destruidas), es decir, aquellas tareas que pueden ser ejecutadas por el scheduler
- Pos_validas: El arreglo pos_validas indica, dada una tarea (numeradas de 0 a 7), si esta es valida o no.
- Indice_gdt: La variable indice_gdt indica cual de las 2 entradas de la GDT de TSS utilizados para el cambio de tareas está siendo usado.
- hayPausa: La variable hayPausa es usada para saber si el juego esta en pausa o no. Se inicializa en 0, y se modifica a 1 cuando se presiona la 'P'. Solo se setea en 0 cuando la 'P' es apretada de vuelta.
- huboSyscall: Usamos la variable huboSyscall como flag para que el scheduler sepa que tiene que saltar a la idle. Dicha variable se resetea en el scheduler
- huboDesalojo: La variable huboDesalojo la ultilazamos como flag para saber si vengo de desalojar una tarea, con lo cual hay que saltar a la idle. La variable se
- Info_desalojo: Utilizamos este arreglo para conservar los valores que debemos imprimir de cada tanque al presionar el número que le corresponde por teclado. (Finalmente decidimos no utilizarla, debido a problemas de implementación).

B) Nuestra tarea sched_proximo_indice devuelve el índice de la próxima tarea a ser ejecutada. Nuestra estructura de scheduler se basa en realidad en una función master que coordina el manejo tanto de las gdts, alternando cada una, como las tareas. Nos pareció más simple modularizar las funciones permitiendonos tener un mejor control de lo que va sucediendo y de como se va modificando la estructura. Es importante aclarar que la función sched_master retornara el indice de la gdt a la que se debe saltar y será la interrupción de clock (o syscall) al retornar la que realizará (o no) el salto.

C) La int 0x52 se encargará de realizar un chequeo de cuál es la syscall que corresponde atender. Básicamente conociendo el parámetro de entrada podrá reconocer a que función implementada en C deberá recurrir. En caso de que la Syscall correspondiente a mover devuelva un 0 significa que el tanque fue desalojado por lo que además la función llamará a *mostrar_regs*

D) La interrupción 32 que corresponde al clock recibe un parámetro luego de haber llamado a la función sched_master. En caso del parámetro ser 0 no habrá salto mientras que en cualquier otro caso realizamos el *far jump* al selector obtenido. Utilizamos las etiquetas de selector y offset tal como se explicó en la clase práctica numero 18.

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

8. Observaciones

8.1. Funcionamiento del scheduler

El scheduler se compone de varias estructuras y algoritmos.

Desde la interrupción de reloj y desde la interrupción de sistema se invoca a la función sched_master. Dicha función se encarga de resolver qué hacer a continuación. Puede ser saltar a una tarea, o no. Si no se debe saltar, sched_master devuelve 0. En caso de tener que saltar, puede ser a una tarea de usuario, o a la idle. El salto a la idle sólo se da en los siguientes casos:

- La pausa está activa y la tarea que está corriendo no es la idle.
- Sched_master es invocada desde la interrupcion 0x52.
- Hubo un desalojo.

  Para tener la información necesaria, el scheduler cuenta con las variables *huboSyscall*, *huboDesalojo* y *hayPausa*. Los comportamientos de estas variables están descriptos anteriormente.

  Si no se detectó ninguna de estas situaciones, lo siguiente a corroborar es si quedan tareas válidas para correr. Si quedan tareas válidas sí o sí se va a llamar a la función **sched_cambio_task**, incluso el caso en el que la tarea que está corriendo sea la misma que tendría que correr. Esta situación la prevee **sched_cambio_task** (explicado más adelante).

  Si no hay tareas válidas para poner a correr significa que o bien está corriendo la idle, en cuyo caso **sched_master** devuelve 0 (no jump), o bien se desalojó y se invocó a esta función. Esta última situación es una de las nombradas anteriormente, con lo cual, se salta a la idle.

  La función **sched_cambio_task** es invocada con la precondición de que exista una tarea válida para ser puesta en marcha. El objetivo es preparar las estructuras del scheduler para dar el salto, con lo cual creímos cómodo que devuelva el selector de segmento indicado. Utiliza el arreglo de tss *tareas_tss* descripto anteriormente, usando el mecanismo de swap descripto por la cátedra en el enunciado.

  Esta función se encarga de salvar la tss de la tarea anterior (tarea que se estaba ejecutando antes que la actual), cargar la tss de la tarea a correr y actualizar las variables *tarea_actual*, *tarea_anterior* e *indice_gdt*. Es importante aclarar que esta funcón evalúa el caso en el que la única tarea válida para saltar ya está corriendo, en cuyo caso devuelve 0 (no jump).

  La función **sched_proximo_indice** es bastante sencilla. Tiene la precodición de que existe alguna tarea válida, ya que si esto no se cumple quedaría ciclando sin fin. Devuelve el índice (de 0 a 7) que corresponde al arreglo de las tss.

  Por último nos queda mencionar la función **saltar_idle**, cuyo funcionamiento es similar al de **sched_cambio_task**, con la diferencia de ya sabe a dónde tiene que saltar (pues salta a la idle). También actualiza las variables *tarea_actual*, *tarea_anterior* e *indice_gdt*.

8.2. Funcionalidad del teclado

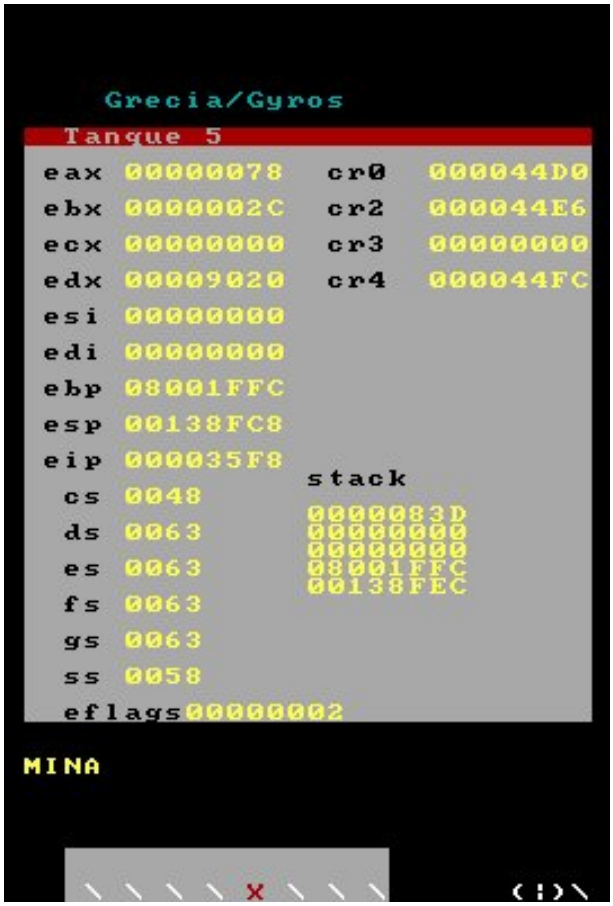
A la hora de implementar la funcionalidad de los números del teclado nos encontramos con varios problemas. Principlamente encontramos que no era posible imprimir correctamente los valores como pretendiamos, pues en la TSS no disponíamos de todo lo necesario. Fue entonces cuando nos propusimos a tomar los valores que podíamos de la TSS y los demás de una estructura auxiliar que iríamos actualizando a medida que fuera necesario.

La funcionalidad parecía estar correctamente implementada, pero nos dimos cuenta que los valores no eran los que pretendíamos mostrar, ya que, si bien corresponden al contexto de la tarea en cuestión, dichos valores son valores de los handlers de las interrupciones (ya sea la de sistema, o las del procesador).

Fué entonces cuando quisimos expandir la estructura de la que disponiamos (que antes solo tenía los cr y la razón de desalojo) para que contenga todo lo que necesitabamos. Es decir, todos los registros y valores de la pila que se tienen que mostrar por pantalla, una escructura grande, pero sencilla, similar a la tss. Esta estructura era actualizada cada vez que se entraba a una interrupción, llamando a una función programada en código ensamblador, que accedía a la estructura y guardaba los valores de los registros, y los valores correctos del *esp, ss, cs, eip y eflags*, que están guardados de una manera particular en la pila (pues en toda interrupción hay un cambio de privilegio, ya que solo una tarea puede caer en una interrupción).

La estructura en sí parecía funcionar, lo mismo con la funcion que guardaba los valores. Pero a la hora de imprimir comprobamos que en la estructura no estaban los valores que habíamos visto antes. Algo escribía sobre nuestra estructura, y no pudimos encontrar qué. Esto no sólo trajo problemas porque no son los valores que queríamos mostrar, si no porque la funcion que imprimía cosas por pantallas usaba el **esp** para acceder a memoria, con lo cual si el valor no era correcto, podía generar errores de paginación o protección general.

Planeabamos que esta fuera nuestra última consulta, pero por problemas de tiempo lo postergamos y cuando finalmente pudimos consultarlo con un ayudante, no logramos descubrir fácilmente cuál era el problema. Por la situación planteada la funcionalidad no está implementada así en la entrega final, aunque se pueden ver la estructura *informácion_auxiliar_desalojo* (comentada) y como la actualizamos al desalojar.
Por no poder corregirlo volvimos atrás y la funcionalidad actual imprime los valores acumulados en la TSS. Así se ve implementado:



8.3. Impresiones desconocidas en pantalla

A medida que avanzamos en el TP nos encontramos con la siguiente imagen:

En ella se ven algunos pedazos de clock que se imprimen por pantalla en lugares extraños. Intentamos debuggear el problema junto con un ayudante y logramos encontrar que correspondía al clock de la tarea idle, pero aún con esa información no logramos evitar que aparezca.

Estos (*¿simpaticos?*) elementos se hacen presentes siempre en los mismos lugares y en ciertos casos hasta parecen ser clocks independientes que avanzan cada muchos ciclos de clock.
No creemos que corresponda a un error de gravedad y dado que lo consultamos y ni así logramos encontrar porque sucede desestimamos la idea de dedicar tiempo a corregirlo, ya que, como mencionamos antes, no abundaba.

8.4. El mapa y la memoria

El mapa representa lugares de la memoria, más especificamente páginas de un área del kernel (el área libre). Estos lugares se van mapeando por los tanques, a media que se van moviendo. Los tanques disponen de un método para

moverse que no se relaciona con la memoria en sí, sino que son posiciones relativas a su lugar en el mapa. Es decir, su código se sigue ejecutando donde estaba (el eip es el mismo), pero su representación en el juego cambió.
Para poder facilitarnos el trabajo decidimos tratar el mapa como una serie de puntos que definimos así:

```
typedef struct posicion_en_mapa {
short fila;
short col;
} punto;
```

Con una representación del mapa elegida, debíamos encontrar la manera de representar toda la información que necesitabamos para el correcto desarrollo del juego. Esta estructura se actualiza cada vez que una tarea produce una syscal. Entonces, cada punto del mapa se vincula a una instancia de la siguiente estructura:

```
typedef struct info_mapa_posicion{
unsigned char tank_1;
unsigned char tank_2;
unsigned char tank_3;
unsigned char tank_4;
unsigned char tank_5;
unsigned char tank_6;
unsigned char tank_7;
unsigned char tank_8;
unsigned char mina;
int mina_de; // Especifica a quien corresponde la mina.
} info_pos;
```

Gracias a esta estructura podemos saber si un tanque ya paso por cierto lugar o no (nos permite poder pintar correctamente el campo), si hay o no minas y, en caso de haberlas a quién corresponden.
Para facilitarnos el pasaje de punto del mapa a posición de memoria implementamos dos simples funciones que nos permiten pasar de una representación a otra.

```
punto hex2punto(unsigned int dir_fis){
punto res;
unsigned int offset = dir_fis - 0x400000;
unsigned int res_div = offset / (50*4096);
unsigned int resto = offset - (res_div * 50*4096);
res.col = resto / 4096;
res.fila = res_div;
return res;
}

unsigned int punto2hex(punto pos){
return ( 0x400000 + pos.fila * (4096*50) + pos.col * (4096));
}
```