Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico III

System Programming

Grupo: Tú me pixeleas

Integrante	LU	Correo electrónico
Costa, Manuel José Joaquín	035/14	manuc94@hotmail.com
Gatti, Mathias Nicolás	477/14	mathigatti@gmail.com
Pondal, Iván	078/14	ivan.pondal@gmail.com

Índice

1. Introducción

El trabajo presentado consiste en la elaboración de un sistema mínimo mediante el cual se profundizaron los conceptos estudiados de *System Programming*. Para esto se siguieron una serie de ejercicios por los cuales en forma gradual se fue construyendo el producto final.

De forma análoga a la resolución de los ejercicios, la estructura del informe seguirá la paso por paso el desarrollo describiendo cada sección junto a la explicación de las decisiones que consideramos más importantes para el correcto funcionamiento del trabajo. A grandes rasgos, contaremos con un pasaje a modo protegido, configuración de segmentación e interrupciones, paginación, manejo de tareas y un scheduler.

El sistema resultante es un juego dentro del cual los jugadores deben soltar perros que buscan huesos y los llevan a su cucha, otorgándole así puntos al jugador correspondiente. El jugador con más huesos será el vencedor.

Para el desarrollo del mismo contamos con *Bochs* que nos permitió emular la computadora que cargaba y ejecutaba nuestro sistema, y en lo que respecta lenguajes de programación se utilizaron C y ASM alternándolos según la rutina que estuviéramos desarrollando.

2. Desarrollo

2.1. Pasaje a modo protegido y segmentación

En esta sección explicaremos como realizamos la creación de la *Global Descriptor Table* (GDT) y como, una vez hecho esto, pasamos a modo protegido.

2.1.1. Inicialización de la GDT

Del código que viene dado se deben agregar en principio cinco segmentos (Para la parte de tareas tuvimos que agregar algunos mas los cuales serán explicados en breve), estos estan conformados por dos de datos, dos de código y uno de la pantalla de video, con niveles de privilegio de 0 o 3.

Para esto seteamos cada segmento, a partir del índice 8 de la gdt por restricciones del trabajo práctico. Las posiciones anteriores a la 8 quedan en 0. Los cuatro primeros segmentos direccionan los primeros 500 MB de memoria, desde la posición 0 hasta el megabyte número 500, en ese espacio entran 128.000 (0x1F400) bloques de 4 KB por lo tanto el límite será 0x1F3FF ya que contamos desde el 0, los de código son del tipo 10 (0x0A) y los de datos del tipo (0x01), el límite es (0xF3FF) y el nivel de privilegio 0 o 3 según corresponda. La granularidad esta activada (0x01). Los bits P, L, D/B y AVL se ponen en 1, 0, 1 y 0 respectivamente para todas las entradas.

Al crear la parte de tareas del sistema tuvimos que agregar 18 segmentos nuevos a la GDT, estos seran los correspondientes a las TSS's de tarea_inicial, IDLE, los 8 perros posibles que puede tener el jugador A simultaneamente y lo mismo para el jugador B (Hablaremos mas de esto en la parte de tareas del informe). Estos segmentos van del número 13 al 30

Para configurar estos segmentos de la GDT se pone como base el 0, esto será luego seteado correctamente por la función *tss_inicializar* la cual pone la dirección de la ubicación de la TSS correspondiente, cada segmento tendrá como limite 104 Bytes (0x0067) para que la GDT interprete eso como bytes ponemos la granularidad en 0. El nivel de privilegio para las tareas de los jugadores será de 3. Las tareas que serán manejadas unicamente por el sistema, *tarea_inicial* y IDLE tendran el DPL en 0. El bit de present va en 1 para todos los segmentos y todo lo demás quedará en 0.

El funcionamiento de *tss_inicializar* es muy simple, hay 4 variables globales en *tss.c* las cuales son tss_inicial, tss_idle, tss_jugadorA y tss_jugadorB, los segmentos deberían tener como base estas direcciones por lo que el objetivo de esta función es setearlos correctamente, para eso utilizamos un macro definido en *tss.h* llamado TSS_PERRO_ENTRY el cual basicamente reconoce a partir de sus parametros que entrada de la gdt se desea setear y luego se pone como base la dirección del perro correspondiente segun a cual jugador corresponda.

```
void tss_inicializar() {
        gdt[GDT_ID13_TSS_INICIAL_DESC].base_0_15 =
        (unsigned short)((uint)&tss_inicial & 0xFFFF);
        gdt[GDT_ID13_TSS_INICIAL_DESC].base_23_16 =
        (unsigned char)(((uint)&tss_inicial >> 16) & 0xFF);
        gdt[GDT_ID13_TSS_INICIAL_DESC].base_31_24 =
        (unsigned char)(((uint)&tss_inicial >> 24) & 0xFF);

        gdt[GDT_ID14_TSS_IDLE_DESC].base_0_15 =
        (unsigned short)((uint)&tss_idle & 0xFFFF);
        gdt[GDT_ID14_TSS_IDLE_DESC].base_23_16 =
        (unsigned char)(((uint)&tss_idle >> 16) & 0xFF);
        gdt[GDT_ID14_TSS_IDLE_DESC].base_31_24 =
        (unsigned char)(((uint)&tss_idle >> 24) & 0xFF);

        TSS_PERRO_ENTRY(15, A, 0);
        TSS_PERRO_ENTRY(16, A, 1);
        TSS_PERRO_ENTRY(17, A, 2);
        TSS_PERRO_ENTRY(18, A, 3);
        TSS_PERRO_ENTRY(19, A, 4);
        TSS_PERRO_ENTRY(20, A, 5);
        TSS_PERRO_ENTRY(21, A, 6);
        TSS_PERRO_ENTRY(22, A, 7);
```

```
TSS_PERRO_ENTRY(23, B, 0);
TSS_PERRO_ENTRY(24, B, 1);
TSS_PERRO_ENTRY(25, B, 2);
TSS_PERRO_ENTRY(26, B, 3);
TSS_PERRO_ENTRY(27, B, 4);
TSS_PERRO_ENTRY(28, B, 5);
TSS_PERRO_ENTRY(29, B, 6);
TSS_PERRO_ENTRY(30, B, 7);

}
```

2.1.2. Pasaje a modo protegido

Antes de pasar a modo protegido el sistema operativo necesita saber donde esta la gdt y cual es su tamaño para esto cargamos a traves de lgdt el registro GDTR, el cual tiene esta información. También habilitamos A20 para permitir el acceso a direcciones superiores a 2^{20} bits.

Luego realizamos un salto con la instrucción *jmp 0x40:modoprotegido* para poner a cs en el valor correcto, el 0x40 es para posicionarnos en el segmento de código de nivel 0 (indice 8 en la gdt). Una vez hecho esto estamos listos para pasar a modo protegido poniendo el bit PE de cr0 en 1.

Luego debemos establecer los selectores de segmentos de datos de nivel 0 y la base de la pila la cual se requirió que estuviera en la posición 0x27000

```
mov ax, 0x50
mov ds, ax
mov es, ax
mov gs, ax
mov fs, ax
mov ss, ax
mov ebp, 0x27000
mov esp, ebp
```

Hay un segmento mas que seteamos con el único objetivo de verificar el buen funcionamiento de la segmentación, este es *GDT_ID12_SCREEN_DESC*, en la función *pintar_esquina_superior_izquierda* definida en *kernelasm* se puede ver como accedemos al segmento y luego reescribimos los dos primeros pixels de la pantalla

```
pintar_esquina_superior_izquierda:
mov ax, 0x60    ; Offset de SCREEN
mov ds, ax
mov byte [0x0], 'X'
mov byte [0x1], 0x4
mov ax, 0x50    ; Offset de KERNEL_DATA
mov ds, ax
ret
```

2.2. Excepciones

En esta sección explicaremos como realizamos la creación de la *Interrupt Descriptors Table* (IDT) y su posterior carga en el *kernel*, así como también en qué consisten las rutinas de atención.

2.2.1. Inicialización de la IDT

La IDT es un arreglo de tamaño 255 de `idt_entry`, la cual posee los campos a setear de una *interrupt gate*. Como en este apartado solo nos interesa asociar las excepciones internas del procesador, únicamente generamos las primeras 20 entradas de la tabla, siguiendo la tabla 6-1 del tercer volumen del manual de Intel [1]. Para inicializarla, utilizamos la función `idt_inicializar` definida en `idt.c`, y cuya implementación se basa en llamar 20 veces al macro `IDT_ENTRY`, definido en el mismo archivo, con los respectivos números de la excepción y el privilegio del descriptor (que en este caso querremos que sea siempre 0, pues es el *kernel* quien manejará estas excepciones).

[1] Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3: System programming guide, Chapter 6

La principal tarea de esta sección es, entonces, configurar correctamente la macro `IDT_ENTRY`. También tendremos definidas 20 rutinas de atención para cada excepción, que en 2.2.3 explicaremos detalladamente. En la figura (1) mostramos como seteamos la entrada de la IDT correspondiente a la excepción n del procesador.

31 — 16	15	14 — 13	12		8	7 — 5	4 — 0
`&_isrn`[31:16]	1	00	0	1	110	000	00000
Offset (parte alta)	P	DPL	M	D	Interrupt gate	M	Reserved

31 — 16	0
0x40	`&_isrn`[15:0]
Segment Selector	Offset (parte baja)

Figura 1: Interrupt gate correspondiente a la excepción n del procesador. `_isrn` es la rutina de atención de la interrupción n, mientras que &`_isrn` es la dirección donde se encuentra alojada la misma; el bit D en 1 indica que el gate es de 32bits. La M implica que el manual pide que esos bits estén seteados de esa forma.

Es importante destacar que el selector de segmento 0x40 corresponde a un segmento de código, pues recordemos que lo que esperamos encontrar en la dirección 0x40:`&_isrn` es el código de un *handler* de interrupción. Además, el segmento requiere privilegio de *kernel*, lo cual es razonable pues esperamos que sea este quien resuelva las excepciones.

2.2.2. Carga de la IDT

Para cargar el descriptor de la IDT (compuesto por su posición base y su tamaño) al IDTR usamos la operación `lidt` de la siguiente forma desde `kernel.asm`

```
lidt [IDT_DESC]
```

2.2.3. Rutina de atención de las excepciones

La rutina de atención de cada una de las 20 excepciones (que esencialmente resultan iguales) se encargan de desalojar a la tarea perro que produjo la excepción [2] y saltar a la tarea idle (cuyo selector de tss es el 0x70), hasta que acabe el *quantum*. Esta funcionalidad se implementa en las últimas líneas de la rutina.

Al entrar, lo primero que se hace es verificar si se está en modo de debug o no. De no estarlo, simplemente se salta al final y se procede de la forma descripta arriba. Caso contrario, se guardan, en la estructura `debug_info`, los valores de todos los registros y flags de la tarea al momento de producirse la excepción. Es importante notar que al estar en la rutina de atención los valores de eip, esp, ss, cs y los flags, son del kernel y no de la tarea. No obstante como los mismos están guardados en la pila podemos recuperarlos de ahí. Luego llamamos a la función `screen_pantalla_debug` que se encarga de setear `debug_screen_on` e imprimir la pantalla correspondiente. Finalmente, se continua con la rutina de desalojo normal y se salta a idle.

2.3. Paginación básica

Aquí contaremos los pasos necesarios para poder activar la paginación y que esta esté configurada para realizar *identity mapping*. Este término se utiliza para describir un direccionamiento de memoria donde la dirección virtual coincide uno a uno con la física.

Las páginas que definimos nos mapean las direcciones `0x00000000` a `0x003FFFFF`, el directorio de páginas se encuentra en la dirección `0x27000` y la tabla de páginas en `0x28000`.

2.3.1. Rutinas de mapeado y desmapeado

Escribimos dos rutinas para realizar la tarea, la de mapeado que nos permite asociar una dirección virtual con una física, y la de desmapeado que dada una dirección virtual la desasocia de memoria.

[2] Más precisamente lo marca como libre y luego en la siguiente interrupción de reloj se desaloja efectivamente del scheduler, a menos que se estuviera en modo de debug. En el último caso se desalorá luego de salir de dicho modo.

Mapeado

La función `mmu_mapear_pagina` toma como parámetros la dirección `virtual`, el contenido del registro `cr3`, la dirección `física` y por último los atributos `attrs`.

1. Extraemos la dirección del directorio de tablas de `cr3`.

```
base_directorio_tablas = 0xFFFFF000 AND cr3
```

2. Calculamos el offset dentro del directorio de tablas en base a `virtual`.

```
offset_directorio_tablas = ((0xFFC00000 AND virtual) >> 22) << 2
```

3. Obtenemos el Page Directory Entry correspondiente.

```
pde = *(base_directorio_tablas + offset_directorio_tablas)
```

4. Extraemos la dirección del directorio de páginas de `pde`.

```
base_directorio_paginas = 0xFFFFF000 AND pde
```

5. Calculamos el offset dentro del directorio de páginas en base a `virtual`.

```
offset_directorio_paginas = ((0x003FF000 AND virtual) >> 12) << 2
```

6. Obtenemos el puntero al Page Table Entry correspondiente.

```
ptr_pte = base_directorio_paginas + offset_directorio_paginas
```

7. Asignamos al Page Table Entry la dirección de la `física` y los atributos `attrs` y forzamos que se limpien las tablas cacheadas para evitar la posibilidad de estar leyendo una entrada desactualizada.

```
*ptr_pte = (0xFFFFF000 AND fisica) OR attrs
Limpio cache de tablas
```

Desmapeado

La función `mmu_unmapear_pagina` toma como parámetros la dirección `virtual` y el contenido del registro `cr3`. Los primeros 6 pasos serán idénticos a los de `mmu_mapear_pagina`.

7. Seteamos en el Page Table Entry el bit de `present` en 0 y limpiamos las tablas cacheadas para reflejar los cambios.

```
*ptr_pte = *ptr_pte AND 0xFFFFFFFE
Limpio cache de tablas
```

2.3.2. Generación del directorio de páginas

Para mapear las direcciones solicitadas, desarrollamos la función `mmu_inicializar_dir_kernel` que se encarga de generar la primer entrada en el directorio de tablas y su tabla de páginas asociada.

1. Primero creamos nuestro Page Directory Entry en la dirección `0x27000`.

```
ptr_pd = 0x27000
*ptr_pd = 0 // Limpiamos la entrada
*ptr_pd = *ptr_pd OR 0x3 // Asignamos atributos
```

   El número `0x3` equivale en binario a `11`. Estos dos primeros bits encendidos corresponden dentro del Page Directory Entry a que el directorio de páginas estará presente y que además podrá ser leído y escrito.

```
*ptr_pd = *ptr_pd OR 0x28000 // Asignamos direccion
```

2. Llenamos el directorio de páginas aplicándole a cada entrada los mismos atributos descritos en el punto anterior.

```
Para i de 0 a 1024
        direccion_pagina = i << 12
        mmu_mapear_pagina(direccion_pagina, 0x27000, direccion_pagina, 0x3)
Fin para
```

2.3.3. Activación de la paginación

Finalmente, teniendo las rutinas explicadas previamente, procedimos a activar la paginación en el procesador. El código siguiente corresponde a una sección del kernel.

1. Llamamos a nuestra rutina de inicialización del directorio de tablas

```
call mmu_inicializar_dir_kernel
```

2. Cargamos la dirección del directorio de tablas en el registro `cr3`.

```
mov eax, 0x00027000
mov cr3, eax
```

3. Activamos la paginación encendiendo el bit más significativo del registro `cr0`.

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

2.4. Paginación dinámica

Procedemos a explicar la segunda parte de lo que es la paginación en el sistema. Con paginación dinámica lo que realizamos es un administrador básico de la memoria libre y generamos la rutina que se encarga de construirle el mapeo de memoria a una tarea perro.

2.4.1. Adminsitrador de memoria libre

Para adminsitrar la memoria libre del sistema, necesaria para inicializar las diversas estructuras del mismo, lo que realizamos es un simple contador que comienza en la dirección `0x102000`.

Su funcionamiento es el siguiente, cuando el sistema necesita memoria, pide la siguiente página libre que corresponde al valor actual del contador y luego a este se lo incrementa en `0x1000`, que es el tamaño de una página.

2.4.2. Modificaciones en rutina de mapeado

Dado que ahora tenemos la posibilidad de utilizar memoria dinámica, nuestra rutina de mapeado debió ser actualizada para la eventualidad en que se quisiera mapear una dirección que no se encontrase en nuestro directorio de tablas.

Luego del paso 3 de la rutina de mapeo, realizamos las siguientes operaciones:

```
Si pde AND 0x1 == 0 // Si no esta presente
        nueva_pagina_memoria = mmu_proxima_pagina_fisica_libre
        inicializamos la pagina (la llenamos de cero)
        *ptr_pde = nueva_pagina_memoria
Fin si

*ptr_pde = *ptr_pde OR attrs_pde // Seteamos sus atributos
pde = *ptr_pde // Cargamos nuestro page directory entry
```

2.4.3. Inicialización de memoria para tarea perro

Para crear el mapa de memoria de una tarea perro debemos tener en consideración varias cuestiones:

1. Debe tener su propio directorio de tablas y a su vez tablas de páginas.
2. Debe realizar identity mapping con permisos restringidos al kernel.
3. Deberá tener su código copiado a la dirección física que corresponde a su posición en el mapa del juego en modo de sólo lectura.
4. Tendrá la dirección `0x400000` mapeada en modo lectura/escritura a una página de memoria que compartirá con los perros del mismo jugador (`0x100000` jugador A, `0x101000` jugador B).
5. Tendrá la dirección `0x401000` mapeada en modo lectura/escritura que llevará a la dirección física donde se encuentra su código.

Es así entonces que para lograr esto realizamos la siguiente secuencia de pasos:

1. Generamos nuestro nuevo directorio y realizamos identity mapping.

```
base_directorio_tablas_perro = mmu_proxima_pagina_fisica_libre
base_directorio_paginas_perro = mmu_proxima_pagina_fisica_libre
incializamos ambas paginas (las llenamos de ceros)
incializamos el primer page directory entry asignandole la direccion del
directorio de paginas y atributos de kernel
realizamos identity mapping igual que con el mapa del kernel
```

2. Dependiendo del tipo de perro que se lanzó, mapeamos su código de la dirección correspondiente.

```
mapeamos direccion virtual del codigo en el mapa con la fisica
mapeamos direccion de codigo de tarea con la fisica correspondiente
```

3. Dependiendo del jugador, mapeamos la memoria compartida

```
mapeamos la direccion virtual de memoria compartida con la fisica
correspondiente dependiendo del jugador
```

4. Copiamos el código a la dirección física donde está mapeado

```
mapeamos direccion donde copiar el codigo en mapa de memoria actual
copiamos el codigo a la direccion mapeada
desmapeamos la direccion del mapa de memoria actual
```

2.5. Interrupciones externas

En la sección 2.2, cargamos las entradas de la IDT y generamos las rutinas de atención correspondientes a las excepciones internas del procesador. A continuación, haremos lo propio para las tres interrupciones externas que podrá recibir nuestro sistema: el reloj, el teclado, y una rutina encargada de proveer los servicios del sistema. Las dos primeras serán requeridas por el PIC, mientras que la última será para uso del usuario.

Dado que por defecto el mapeado del PIC entra en conflicto con las excepciones del procesador de la 8 a la 15, es necesario hacer previamente un remapeo. Para eso, la cátedra ya proporciona la función `resetear_pic` que se encarga de remapear las *interrupt's requests* (IRQs) de reloj y de teclado a las interrupciones 32 (0x20) y 33 (0x21) respectivamente. Basta entonces llamar a esta función desde el kernel y luego reactivar el PIC con la función `habilitar_pic`, también provista por la cátedra.

La interrupción para solicitar servicios del sistema será la 0x46.

Las entradas de la IDT se agregarán, al igual que antes, con el macro `IDT_ENTRY`. Las de reloj y teclado tendrán DPL de kernel (0), mientras que para la 0x46 será de usuario (3).

A continuación explicamos que es lo que hace cada una de las rutinas de atención.

2.5.1. Reloj

Esencialmente, el código de esta rutina es el mismo que el sugerido en la clase práctica de scheduler. La única diferencia real es que se agregan algunas líneas usadas para implementar el modo debug.

Como primer paso, tras salvaguardar todos los registros para mantener la transparencia de la interrupción, llamamos a la función `fin_intr_pic1` que se encarga de avisarle al PIC que el IRQ del reloj ya ha sido atendido, y por lo tanto el PIC nuevamente está disponible para recibir IRQs.

Luego se hace la comparación

`cmp dword [debug_screen_on], 1`

para ver si la pantalla especial que salta cuando se produce una excepción en modo debug está activada. En caso afirmativo, simplemente se salta al final de la rutina, sin actualizar el juego ni intercambiar tareas. Esto significa que se seguirá ejecutando la tarea idle hasta que hasta que `debug_screen_on == 0`.

A continuación se llama a la función `sched_atender_tick` [3] la cual se encargará de actualizar todo lo que sea necesario respecto del juego: posición del perro que está corriendo actualmente; removerlo del scheduler en caso de que deje de estar libre, ya sea porque volvió a la cucha y descargó todos sus huesos, o bien porque provocó una excepción; y finalmente setear la siguiente tarea a ejecutar, devolviendo el índice de su TSS en la GDT como output.

Como dicho output se recibe por el registro ax, se lo compara contra cx (al que previamente le cargamos el valor actual del *task register* con `str`) y en caso de ser iguales simplemente se sigue ejecutando la misma tarea. Caso contrario debe realizarse el *task switching*, haciendo un jump far con el selector de segmento correspondiente al tss de la siguiente tarea.

Finalmente, restauramos los registros con los valores que tenían antes de la interrupción.

2.5.2. Teclado

Nuevamente salvamos los valores de todos los registros y llamamos a `fin_intr_pic1`. Luego limpiamos eax, y leemos el *scan code* que se encuentra en el puerto 0x60. Llamamos a la función `game_atender_teclado` pasándole como parámetro dicho *scan code*.

Esta función lo que hará es considerar dos posibles situaciones: una en la que el juego se encuentra pausado pues se ha cometido una excepción en modo debug, provocando que se active la pantalla especial, y por lo tanto la única tecla que tiene validez es la "y", que permitirá salir de dicha pantalla. La otra es que el juego este en modo normal y por lo tanto todas las teclas asociadas a alguna función del juego (según se señala en el enunciado) activan la misma al ser presionadas.

Al volver de `game_atender_teclado`, se restaurán los registros y se finaliza la rutina.

2.5.3. Servicios del sistema

Esta función toma hasta dos parámetros por registro. Por eax siempre recibe el tipo de pedido requerido (moverse, cavar, olfatear, recibir orden). En caso de que el servicio solicitado sea mover al perro, se tomará un parámetro adicional por ecx con alguno de los cuatro códigos de dirección válidos. También para recibir orden se tomará un parámetro más, para determinar de que jugador se desea oir la última orden.

Lo primero que hace la rutina, antes del `pushad` es pushear eax. En un instante veremos la justificación de esto. Se llama a `game_syscall_manejar` pasándole los dos parámetros recibidos (en caso de que el servicio sea cavar u olfatear en ecx habrá basura pero se ignorará completamente). Esta función no hace más que realizar un switch para determinar cuál será la función que resuelva efectivamente el pedido. Dichas funciones ya venían dadas por la cátedra, salvo `game_perro_recibir_orden` cuya implementación, como puede verse, es muy simple.

Al retornar a la rutina de atención, si el servicio solicitado era olfatear o recibir orden, entonces se retornará un valor por eax. Aquí es donde entra en juego el `push eax` que habíamos hecho al principio de todo. Como `mete` 8 registros de 4 bytes cada uno en la pila, sabemos que nuestro primer `push` está en esp+32. Copiamos el actual valor de eax ahí. La razón por la que hacemos esto es que al realizar el `popad` se pisa el valor de eax, perdiendo el resultado de la función. Pero por lo que hicimos, simplemente hace falta realizar un `pop eax` al finalizar la rutina para recuperarlo.

2.6. Tareas

Para poder manejar las tareas de forma dinámica, permitiendo que el procesador pase de una a otra sin perder el contexto de trabajo que tenía la tarea anterior es que se utilizan las TSS (*task state segment*. En esta sección explicaremos brevemente como configuramos todo para su buen funcionamiento.

[3] La explicación correspondiente a la implementación de esta función se desarrolla en la sección del scheduler.

2.6.1. Entradas de la GDT

Cada TSS, tiene un descriptor que se declara en la gdt, esto fue explicado previamente en la sección correspondiente. Veamos un poco que papel tendrá cada uno de esos segmentos. La tarea inicial, será creada para tener algo a lo que saltar al ir a la primer tarea, es necesaria ya que el procesador siempre intentara guardar el contexto de las tareas si al arrancar no hay nada las cosas irán mal. La TSS de esta tarea puede ser llenada de la forma que sea ya que no vamos a prestar atención a su contexto, la descartaremos tan pronto sigamos con una nueva tarea. Por otro lado teniamos la TSS de la tarea IDLE, esta será fundamental para cubrir los huecos de tiempo en los que el sistema no tenga nada que hacer.

Dicho esto pasemos a ver como se inicializa *tss_inicializar_idle*

```
void tss_inicializar_idle() {
        tss_idle.ptl = 0;
    tss_idle.unused0 = 0;
    tss_idle.esp0 = 0x27000;
    tss_idle.ss0 = 0x50;
    tss_idle.unused1 = 0;
    tss_idle.esp1 = 0;
    tss_idle.ss1 = 0;
    tss_idle.unused2 = 0;
    tss_idle.esp2 = 0;
    tss_idle.ss2 = 0;
    tss_idle.unused3 = 0;
    tss_idle.cr3 = 0x27000;
    tss_idle.eip = 0x16000;
    tss_idle.eflags = 0x00000202;
    tss_idle.eax = 0;
    tss_idle.ecx = 0;
    tss_idle.edx = 0;
    tss_idle.ebx = 0;
    tss_idle.esp = 0x27000;
    tss_idle.ebp = 0x27000;
    tss_idle.esi = 0;
    tss_idle.edi = 0;
    tss_idle.es = 0x50;
    tss_idle.unused4 = 0;
    tss_idle.cs = 0x40;
    tss_idle.unused5 = 0;
    tss_idle.ss = 0x50;
    tss_idle.unused6 = 0;
    tss_idle.ds = 0x50;
    tss_idle.unused7 = 0;
    tss_idle.fs = 0x50;
    tss_idle.unused8 = 0;
    tss_idle.gs = 0x50;
    tss_idle.unused9 = 0;
    tss_idle.ldt = 0;
    tss_idle.unused10 = 0;
    tss_idle.dtrap = 0;
    tss_idle.iomap = 0xFFFF;
}
```

Para la configuración de las opciones de la TSS IDLE pusimos la dirección de la pila en el mismo lugar que la del kernel (0x27000), aunque las pilas de menores privilegios, esp1 y esp2 estan en 0, cr3 vale igual que el cr3 del kernel, iomap vale 0xFFFF, y los selectores de segmentos se pusieron segun correspondian. En eip ponemos la dirección de la tarea la cual se aclara en la consigna que es la 0x16000.

Una vez que esta todo configurado, inicializamos las tss y cargamos la tarea inicial en kernel.asm

```
        ; Cargar tarea inicial
```

```
        mov ax, 0x68
        ltr ax
```

2.6.2. Construcción de Tareas

Para la construcción de la tss de los perros de manera similar que lo que hicimos con tss_idle creamos la siguiente función.

```
void tss_construir_tarea(perro_t *perro, int index_jugador, int index_tipo){
        uint cr3 = mmu_inicializar_memoria_perro(perro, index_jugador, index_tipo);
        uint ebp = CODIGO_BASE + 0xFFF;
        uint esp = ebp - 12;

        tss* ptr_tss = (index_jugador == JUGADOR_A) ?
         &tss_jugadorA[perro->index] :
         &tss_jugadorB[perro->index];

        uint pila_nivel_0 = mmu_proxima_pagina_fisica_libre() + 0xFFF;

        ptr_tss->ptl = 0;
        ptr_tss->unused0 = 0;
        ptr_tss->esp0 = pila_nivel_0;
        ptr_tss->ss0 = 0x50;
        ptr_tss->unused1 = 0;
        ptr_tss->esp1 = 0;
        ptr_tss->ss1 = 0;
        ptr_tss->unused2 = 0;
        ptr_tss->esp2 = 0;
        ptr_tss->ss2 = 0;
        ptr_tss->unused3 = 0;
        ptr_tss->cr3 = cr3;
        ptr_tss->eip = CODIGO_BASE;
        ptr_tss->eflags = 0x00000202;
        ptr_tss->eax = 0;
        ptr_tss->ecx = 0;
        ptr_tss->edx = 0;
        ptr_tss->ebx = 0;
        ptr_tss->esp = esp;
        ptr_tss->ebp = ebp;
        ptr_tss->esi = 0;
        ptr_tss->edi = 0;
        ptr_tss->es = 0x5B;
        ptr_tss->unused4 = 0;
        ptr_tss->cs = 0x4B;
        ptr_tss->unused5 = 0;
        ptr_tss->ss = 0x5B;
        ptr_tss->unused6 = 0;
        ptr_tss->ds = 0x5B;
        ptr_tss->unused7 = 0;
        ptr_tss->fs = 0x5B;
        ptr_tss->unused8 = 0;
        ptr_tss->gs = 0x5B;
        ptr_tss->unused9 = 0;
        ptr_tss->ldt = 0;
        ptr_tss->unused10 = 0;
        ptr_tss->dtrap = 0;
        ptr_tss->iomap = 0xFFFF;
```

}

Como cosas que valen la pena resaltar podemos ver que para conseguir una pila de nivel 0 nueva debemos pedir una nueva página utilizando la función explicada en la sección anterior y setear el inicio de la fila al final de dicha página; para esp debemos restar 12 al valor del inicio de la pila ya que están apilados los 2 argumentos de las tareas y la dirección de retorno.

2.6.3. La Interrupción 0x46

En nuestro sistema operativo la interrupción 0x46 es la rutina de atención de servicios, es ejecutada por los jugadores para decidir las acciones de los perros. Estas pueden variar entre *game_perro_mover*, *game_perro_cavar*, *game_perro_olfatear* y *game_perro_recibir_orden* todos estos métodos estan definidos en *perro.c* por la catedra. Una vez hecha la acción si todavia el quantum no se acabo se llama a la tarea idle para que esta espere hasta el turno del siguiente jugador.

2.7. Scheduler

El scheduler cumple un rol fundamental ya que es el encargado de permitir que nuestro sistema sea multitarea, otrogándole a cada una un tiempo determinado para que corran en el mismo. Para nuestro juego contamos con un máximo de 17 tareas donde tenemos 16 para los perros y una para la tarea idle.

Cada jugador puede tener hasta 8 tareas perros corriendo en simultáneo, con lo cual todas las decisiones tomadas para la elaboración del manejo de tareas fue teniendo en consideración esta cota.

2.7.1. Rutina para ejecutar siguiente tarea

Esta es quizás la rutina más importante de esta sección, ya que es la que decide qué tarea viene después de la actual. Para esto antes procedemos a explicar las estructuras que nos permiten llevar a cabo esta operación.

Por un lado tenemos la estructura `sched_task_t` que posée los campos `perro` y `gdt_index`. Esta estructura contendrá para cada tarea presente en el scheduler su perro asociado y el offset en la gdt que apunta a su TSS correspondiente. A su vez tenemos `sched_t` compuesto por un arreglo `tasks` de 17 `sched_task_t` y un entero `current` conteniendo el índice de la tarea actual.

Ahora, la última posición del arreglo `tasks` está ocupada por la tarea idle, que contendrá su offset en la gdt correspondiente pero la variable `perro` estará en NULL. Con el resto de las posiciones, $(0, \ldots, 15)$, tendremos que las pares corresponderán al jugador A y las impares al jugador B.

De esta manera, la rutina que busca la siguiente tarea se comporta de la siguiente manera:

1. Incializamos variables.

```
encontre_proximo = FALSE
i = scheduler.current
tarea_actual = scheduler.tasks[i]
tarea_siguiente = NULL
```

2. Ciclamos dentro del arreglo `tasks`.

```
Mientras no encontre_proximo
```

   a) Incremento mi variable i teniendo en cuenta que puede darle la vuelta al arreglo y cargo la posible tarea siguiente.

```
i = (i + 1) MODULO 17
tarea_siguiente = scheduler.tasks[i]
```

   b) Considero el caso en que haya recorrido todo el arreglo hasta volver a la posición de la tarea actual.

```
Si i == scheduler.current
        Si tarea_actual.gdt_index == NULL // Tarea muerta o idle
                i = 17 // Mando a la tarea idle
        Fin Si
        encontre_proximo = TRUE
```

c) Y ahora para el caso en que no di toda la vuelta.

```
                Si no
                        Si tarea_siguiente.gdt_index != NULL AND
                                        tarea_siguiente.perro != NULL
                                encontre_proximo = TRUE
                        Fin Si
                Fin Si
        Fin Mientras
```

3. Devolvemos el índice de la próxima tarea a ejecutar.

```
Devuelvo i
```

2.7.2. Rutinas para agregar y remover tareas

Teniendo lo visto en la entrada anterior, las rutinas para agregar y remover tareas son bastante simples.

Agregar tarea

Para agregar una tarea, el scheduler busca un slot libre dentro del arreglo `tasks` que le pertenezca al jugador dueño de la tarea (Si es el jugador A un slot par, si es el B, uno impar). Para esto se recorren las posiciones válidas para el jugador designado hasta encontra una cuyo `gdt_index` sea NULL, que va a suceder cuando este slot pueda utilizarse.

A continuación lo único que se realiza es en esa posición asignar el offset de la gdt de la tarea a ejecutar y el perro correspondiente.

Remover tarea

Para remover una tarea, dado como parámetro el `gdt_index` se busca en el arreglo `tasks` su tarea asociada y luego se setean tanto su offset en la gdt como el perro en NULL.

2.7.3. Rutina para atender un tick

Esta rutina también tiene mucha importancia ya que es la que se llama en cada tick de reloj, donde debe cambiar la tarea actual por la siguiente a ejecutar.

1. Comenzaremos por llamar a la rutina que se encarga de actualizar parámetros del juego.

```
game_atender_tick(game_perro_actual)
```

2. Procedemos viendo si es necesario remover la tarea actual

```
Si game_perro_actual != NULL AND game_perro_actual->libre == TRUE
        sched_remover_tarea(scheduler.tasks[scheduler.current].gdt_index)
```

3. Buscamos la tarea a ejecutar y actualizamos la tarea y perro actuales

```
scheduler.current = sched_proximo_a_ejecutar
game_perro_actual = scheduler.tasks[scheduler.current].perro
```

4. Finalmente devolvemos el offset en la gdt para que se realice el salto a la tarea.

```
Devolver scheculer.tasks[scheduler.current].gdt_index
```

2.8. Modo Debug

Para armar el *Modo Debug* tuvimos que alterar un poco el funcionamiento de las rutinas de atención. Esto esta detallado en la sección de interrupciones pero contaremos aquí lo relevante de las interrupciones para el funcionamiento del modo debug. La rutina del teclado, al llamar a *game_atender_teclado* reconoce la letra *y*, la cual al ser presionada llama a *game_jugador_debug*, es importante notar que si la pantalla del *Modo Debug* esta activada ningún jugador deberia poder realizar algún tipo de acción para esto se agrego un condicional a *game_atender_teclado*. El método al cual se llama al presionar la letra *y* esta descrito en *game.c*.

```
void game_jugador_debug(){
                if(debug_screen_on == TRUE){
                        debug_screen_on = FALSE;
                        screen_restore_backup();
                }
                else{
                        if(debug_on == FALSE){
                                debug_on = TRUE;
                                print("MODO DEBUG ACTIVADO", 55, 0, 0x0f0f);
                        }
                        else{
                                debug_on = FALSE;
                                screen_pintar_rect(0, 0x00, 0, 53, 1, 23);
                        }
                }
}
```

La función utiliza dos variables globales, *debug_on* y *debug_screen_on*, la primera indica si el modo debug esta encendido, la segunda si la pantalla del modo debug esta activada. En el caso de que *debug_screen_on* valga 0 lo único que se hace es imprimir en pantalla que el modo debug esta activado cuando corresponda y cambiar el valor de *debug_on*. En el caso de que la pantalla este activada esta se apaga con el método definido en *screen.h*, *screen_restore_backup*. Una vez hecha la parte del teclado pasamos a lo que va a hacer la rutina de atención de excepciones. Lo primero que hacemos es ver si estamos en modo debug, si no estamos matamos a la tarea directamente, pero si estamos como primer paso debemos utilizar la estructura llamada *debug_info* la cual nos servira de soporte para almacenar todo el contexto en el que se generó la excepción, una vez hecho eso generamos la pantalla del modo debug para que muestre el contexto en el que se genero el problema, una vez hecho esto ya podemos matar a la tarea y saltar a la tarea idle.

```
_isr%1:
cmp dword [debug_on], 0
je .matarTarea

;....
;guardado del contexto en debug:info
;....

call screen_pantalla_debug

.matarTarea:
push dword [game_perro_actual]
call game_perro_termino
pop ax
;Salto a la tarea idle
jmp 0x70:0
```

Una vez hecho esto se mantendra todo el sistema estático hasta que debido a una interrupción del teclado se cambie el valor de *debug_on*, esto será debido a que la rutina de atencion del reloj si detecta que *debug_screen_on* vale 1 no hace nada.

Una vez que se vuelve a presionar la letra *y game_jugador_debug* restaura la pantalla, pone *debug_screen_on* en FALSE y *debug_on* se mantiene en TRUE para mantener al sistema en *modo debug* como se indicaba en

la consigna. Restaurandose todo esto el sistea continua normalmente el juego, al pasar un ciclo del reloj se continuara con la siguiente tarea que corresponda.

3. Conclusiones

Una vez concluído este trabajo, terminamos con un sistema que resultó en un juego donde pudimos aplicar muchos de los conocimientos estudiados en la materia.

Tuvimos que configurar la segmentación, trabajar con paginación, administrar nuestra propia memoria dinámica, desarrollar un scheduler y correr varias tareas para así llegar al objetivo de tener un juego completamente funcional.

Es así como con este desarrollo pudimos tener una idea general de qué es lo que debe realizar un sistema operativo y cómo se encarga de llevar a cabo cada responsabilidad.

Como posible mejora a futuro si interesase profundizar el estudio de alguno de los componentes construidos se podrían realizas versiones más complejas de los mismos, como lo podría ser un administrador de memoria más inteligente o un scheduler sin las limitaciones de tareas actual.