Organización del Computador II

Trabajo Práctico 3

Emilio Almansi
ealmansi@gmail.com

Álvaro Machicado
rednaxela.007@hotmail.com

Miguel Duarte
miguelfeliped@gmail.com

2do cuatrimestre de 2013

Índice

1. Introducción

Se desarrolló un kernel elemental para la arquitectura Intel x86 con soporte básico para las unidades de segmentación y paginación, manejo de interrupciones y cambio automático de tareas por hardware.

El alcance del trabajo incluye múltiples etapas del proceso de arranque de todo sistema operativo, permitiendo apreciar las herramientas que provee el procesador para la administración del sistema. Durante la etapa de incialización, es necesario llevar a cabo distintos pasos como el traspaso de modo real a modo protegido, la habilitación de la unidad de paginación, o el armado de múltiples estructuras requeridas por el procesador para el manejo de interrupciones o de tareas.

Adicionalmente, se implementó un scheduler rudimentario permitiendo realizar la ejecución por tiempo compartido de múltiples tareas definidas al tiempo de compilación, realizando un ciclo de ejecución no lineal guiado por tics del reloj.

2. Modo real y modo protegido

En el sistema implementado se utilizaron dos modos de trabajo de los múltiples que provee la arquitectura Intel x86: el modo real y el modo protegido. Todos los procesadores de dicha arquitectura comienzan a operar en modo real; es decir, funcionando similarmente al procesador Intel 8086. Esto conlleva un tamaño de palabra de 16 bits, una capacidad de direccionamiento de 1MB, nula protección por hardware, entre otras características. Por el contrario, en modo protegido el procesador puede hacer pleno uso de sus recursos; es decir, su completo set de instrucciones, direccionamiento a 4GB de memoria, potentes mecanismos de protección por hardware, y otras ventajas.

2.1. Modo real

Si bien es deseable trabajar en modo protegido por sobre modo real, existe una secuencia de pasos que deben ser realizados previamente a realizar la transición. En primer lugar, es necesario implementar un bootloader en modo real, el cual se encarga de cargar el código inicial del kernel.

Una vez que el bootloader finaliza, es necesario deshabilitar las instrucciones. Este paso es muy importante pues en esta instancia aún no fue definido el manejo de interrupciones, por lo que la primera interrupción de reloj generaría una excepción sin capacidad de ser manejada.

Posteriormente, se requiere la habilitación del gate A20 para permitir direccionamiento a más de 1MB. Es razonable hacerlo en modo real, cuando de todas maneras el límite de direccionamiento es 1MB. Finalmente, previo al paso a modo protegido se habilitó el uso de 16 colores de video. Por defecto el video está configurado para que el bit mas significativo de las casillas de formato represente la propiedad "blink", es decir si el caracter parpadea o no. Para lograr una pantalla más expresiva, se reconfiguró el sistema para utilizar ese bit como parte del descriptor de color.

Estos pasos relacionados a la configuración del hardware fueron realizados en modo real principalmente para alistar el sistema antes de comenzar la organización del kernel.

2.2. Pasaje a modo protegido

Previo al paso a modo protegido, se debe armar la primer estructura necesaria para el funcionamiento del kernel; la Global Descriptor Table (GDT), la cual se discute en más detalle en secciones subsiguientes.

Activar el modo protegido en principio consiste más que en activar el bit menos significativo del registro de control CR0. Sin embargo, en el momento en que se setea ese bit, se altera la manera de interpretar a los registros de segmento. Para el correcto funcionamiento del sistema en esta etapa, lo que se debe hacer entonces es setear los registros de segmento en valores válidos (según la GDT realizada), empezando por el segmento de código (CS), que es el primero que puede producir errores. Para lograr esto, se realiza un jmp far a la siguiente instrucción, con un valor adecuado para el CS.

A continuación se incluye un extracto del código mostrando este proceso.

```
1      MOV     eax, cr0     ; Se trae el contenido de cr0
2      OR      eax, 1       ; Se cambia el bit correspondiente, el resto quedan iguales
3      MOV     cr0, eax     ; Se hace realmente el seteo.
4
5      JMP     0x90:mp      ; La siguiente instruccion debe cambiar el valor del CS, pues sino se
6                           ; va a generar error de proteccion. Para eso se realiza un JMP FAR a
7                           ; la siguiente instruccion
8      BITS 32              ; Con esto se le avisa a NASM que a partir de ahora se espera codigo de 32 bits.
9  mp:                      ; Esta etiqueta es el destino del JMP FAR, es
10                          ; decir, la instruccion siguiente al JMP.
11
12 ; A partir de aca estamos en modo protegido
```

2.3. En modo protegido

Una vez en modo protegido, es importante realizar dos tareas en primera instancia: acomodar el resto de los registros de segmento (el jmp far configura el CS, pero resta configurar todos los demás) y setear la pila correctamente; es decir, sencillamente asignar a ESP y a EBP los valores adecuados para que la pila quede donde se haya reservado espacio para ella (en el caso de este sistema, es la posición de memoria 0x27000, alojando la pila 0x26000 y 0x26FFF).

3. Global Descriptor Table (GDT)

La GDT (*Global Descriptor Table*) es una estructura de datos que la arquitectura Intel x86 utiliza para almacenar distintos descriptores de sistema, como descriptores de segmento ordinarios o descriptores de Task State Segment (TSS). Estos permiten asignar características o propiedades a distintas secciones de la memoria, o permiten identificar la ubicación y atributos de diferentes estructuras necesarias para el funcionamiento del sistema. En la primer entrada de la tabla siempre se asigna un descriptor nulo, por requerimiento del sistema de protección. El procesador produce una excepción automáticamente cuando se intenta acceder a la posición cero de la GDT.

3.1. Descriptores de segmento y Segmentación flat

Los descriptores de segmento delimentan el tamaño y la ubicación de los segmentos de memoria, así como sus propiedades: quién los puede acceder, que clase información alojan (datos o código), entre otras. De esta forma, permiten particionar el espacio de memoria para ser utilizado por diferentes actores y con diferentes utilidades, garantizando que no se viole la integridad del esquema mediante el sistema de protección.

El mecanismo de segmentación optado en este trabajo consiste en una segmentación tipo flat. Este mecanismo consiste en anular virtualmente la protección por segmentación que provee la arquitectura, armando segmentos superpuestos entre sí, ocupando la totalidad de la memoria. Se configuraron cuatro segmentos: dos con nivel de privilegio 0 (máximo nivel de provilegios), y dos con nivel de privilegio 3 (mínimo nivel de privilegios). Para cada nivel se utilizó un segmento de código y uno de datos.

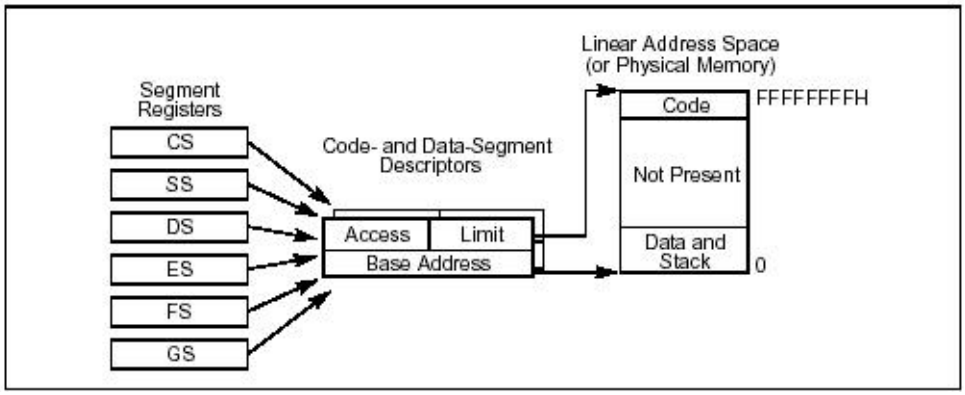

Figura 1: Segmentación flat

La decisión de dejar de lado la protección por hardware brindada a través de la segmentación se basa en la posibilidad de lograr resultados similares, pero de forma más sencilla y flexible, mediante el mecanismo de paginación, descripto en la próxima sección.

Otra consecuencia de la segmentación flat es que los descriptores de segmento con sus parámetros adecuados se conocen de antemano y se configuran en tiempo de compilación. Una vez que el sistema entra en ejecución, estos descriptores de segmento no se vuelven a modificar.

Adicionalmente, se creó un segmento conteniendo la memoria de video, principalmente con fines didácticos ya que el manejo de la pantalla se realizó en su totalidad mediante el segmento de datos de nivel 0.

3.1.1. Atributos descriptores de segmento

Al usar sementación flat, la configuración correcta para los atributos para cada segmento es la siguiente: la base del segmento debe estar en la posición 0, el límite debe ser el tamaño máximo de cada segmento (1,75gb en este caso) menos 1, la granularidad (G) tiene que estar en 1, pues los sectores serán de 4kb, y el bit Present (P) tiene que estar en 1, habilitando el uso del segmento.

El resto de los parámetros varían en cada segmento; los dos segmentos que serán utilizados por el kernel requieren Descriptor Privilege Level (DPL) igual a 0, y los otros dos serán de acceso a nivel usuario con DPL igual a 3. Por último, es necesario especificar el tipo del segmento, ya sea datos o código (uno de cada tipo por cada nivel, como se dijo previamente).

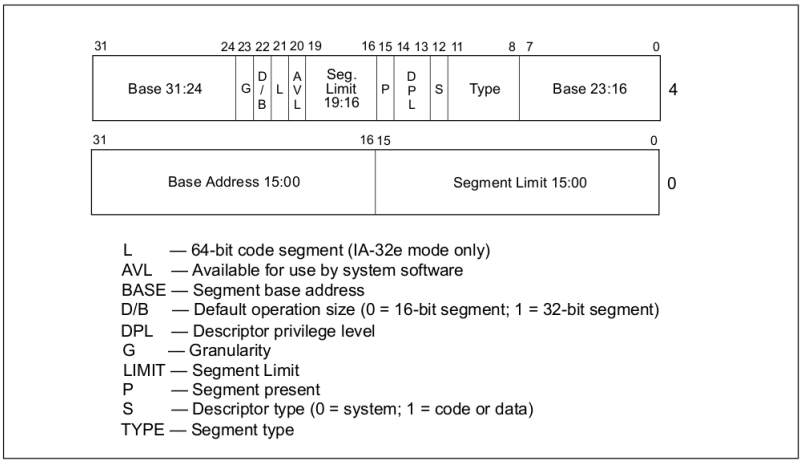

Figura 2: Descriptor de segmento

3.2. Descriptores de TSS

Los descriptores de TSS son necesarios para utilizar el cambio automático entre tareas que provee la arquitectura. Describen la posición y el tamaño de las estructuras donde se almacena el estado de cada tarea del sistema; es decir, el valor de sus registros, la tarea predecesora, y más información.

El almacenamiento de las TSS se realizó de forma dinámica, sin predeterminar su posición en tiempo de compilación. Por eso sus respectivos descriptores deben ser definidos en tiempo de ejecución, lo cual se realizó mediante las siguientes funciones en C:

```
void tss_inicializar_entrada_gdt_tarea_inicial();
void tss_inicializar_entrada_gdt_idle();
void tss_inicializar_entrada_gdt_navio(unsigned int nro_tarea);
void tss_inicializar_entrada_gdt_bandera(unsigned int nro_tarea);
```

Si bien estas funciones tienen mucho en común se las creo por separado por una cuestión pragmática que sirvió para encontrar errores y *bugs* mas fácilmente.

Estas funciones a su vez se engloban todas en otra función escrita en C que inicializa todas las TSS y se llama desde **kernel.asm**

```
void tss_inicializar();
```

4. Paginación

La paginación es un método de direccionamiento de memoria en el cuál se redirigen bloques de direcciones "virtuales" a bloques de direcciones físicas. Estos bloques se llaman páginas, y en la arquitectura Intel en el modo usado en el trabajo tiene un tamaño de 4KB (`0x1000`). La gran ventaja de la paginación es que la unidad en la que se trabaja la memoria (la *página*) tiene siempre el mismo tamaño, lo cuál hace que sea fácil hacer manejos complejos de memoria.

En la arquitectura Intel x86 la paginación se maneja mediante una estructura de sistema en memoria en 2 niveles. El primer nivel se llama *page directory* y el segundo *page table*. Los page directory contienen las direcciones donde se ubican los page table y los page table contiene direcciones físicas de páginas. De esta manera una dirección virtual (es decir la dirección a la que accede una tarea) se interpreta como un índice en el page directory, un índice en la page table obtenida y , finalmente, un offset para la página encontrada en el page table.

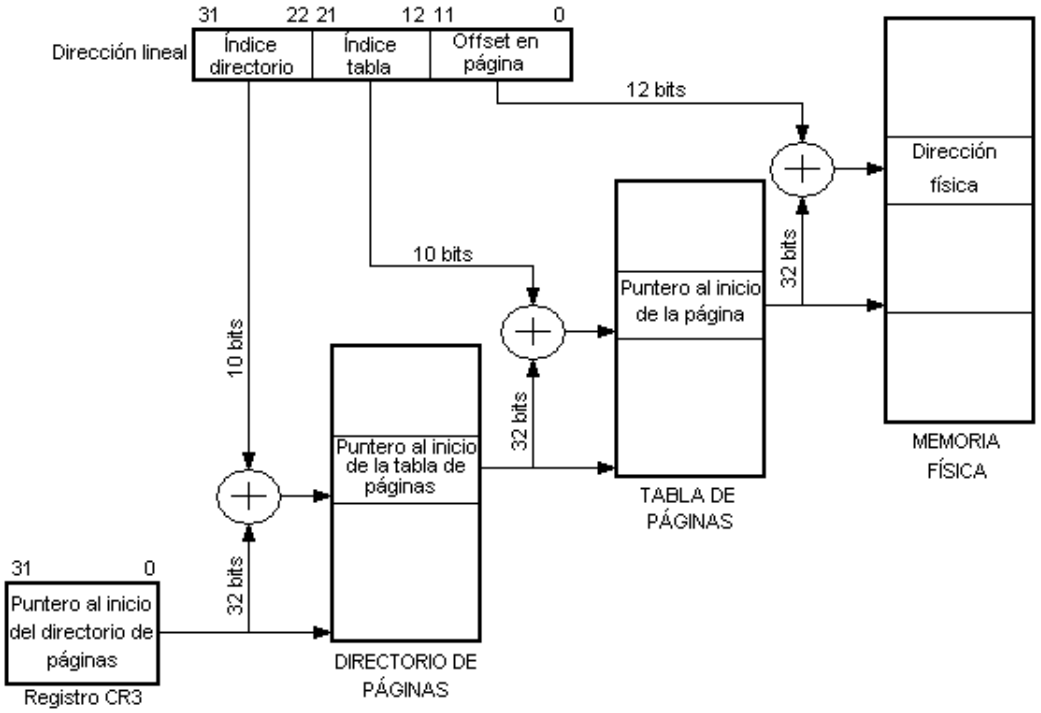

Figura 3: Estructura de paginación

La siguiente ventaja de este mecanismo es que uno puede tener muchas estructuras de paginación en memoria y asignarle una distinta a cada tarea. Esto se hace mediante un registro de control, el **cr3**. Cuando la paginación está activa, cada vez que se realiza un acceso a memoria el procesador usa el **cr3** para encontrar la dirección del page dir. Modificando el valor de **cr3** se puede hacer que diferentes tareas tengan diferentes *mapas de memoria* , es decir que cuando accedan a iguales direcciones virtuales lleguen a diferentes direcciones físicas.

Si bien mas adelante se discute la seguridad de manera específica es importante mencionar que debido a que la segmentación flat anula todos los sistemas de seguridad que provee la segmentación, la paginación debe asegurar toda la seguridad y la integridad de la memoria por sí sola.

Además el mapeo de páginas tiene que permitir que las tareas se ejecuten como si su código estuviera en la posición 1GB (`0x40000000`) y puedan leer (pero no escribir) desde las direcciones `0x40002000-0x40002FFF` direcciones de memoria del kernel.

Para lograr todo esto implementamos un esquema de paginación en el cuál existen páginas compartidas y páginas privadas. Se tuvo especial cuidado con los permisos otorgados a cada tabla para evitar accesos a memoria indeseados.

4.1. Implementación del mapa de memoria

El mapa de memoria se implementó de forma tal que haya identity mapping (la dirección virtual coincide con la física) en los primeros 7.5MB, y después cada tarea tenga los mapeos necesarios para acceder a su código y al ancla a través de las direcciones virtuales a partir de 1GB.

En tiempo de ejecución, se generan todas las estructuras necesarias para que el sistema y las tareas puedan funcionar. Esto incluye un page directory para el kernel y para cada tarea, un page table privado de cada tarea (con permisos de usuario) y 2 page table compartidos por el kernel y las tareas (con permisos de administrador).

Las page table privadas en un principio se inicializan en cero, luego se actualizan de manera adecuada con funciones de mapeo de página descriptas más adelante. En esta instancia, se realizó una copia del código de las tareas de la tierra al mar.

Todo este mecanismo se traduce en un llamado a una función escrita en C, **mmu_inicializar_tareas**, desde kernel.asm.

Todas las tareas deben tener acceso a través de su mapa de memoria a las direcciones de mar y tierra, de forma tal que puedan ser accedidas cuando se ejecuta código de nivel 0 desde el contexto de la tarea, pero no por la tarea misma. Es decir, el kernel y todas las tareas precisan identity mapping de nivel 0 en los primeros 7.5MB, por lo cual las page tables con estas direcciones pueden ser compartidas sin problema. Esto significa todas las primeras entradas de todos los page directories apuntan a la misma page table, redireccionando con permisos de administrador a los primeros 4MB de la memoria física, y todas las segundas entradas de page dir apuntan también a una page table común que direcciona los siguientes 3.5MB. Esta configuración se esquematiza en la siguente figura.

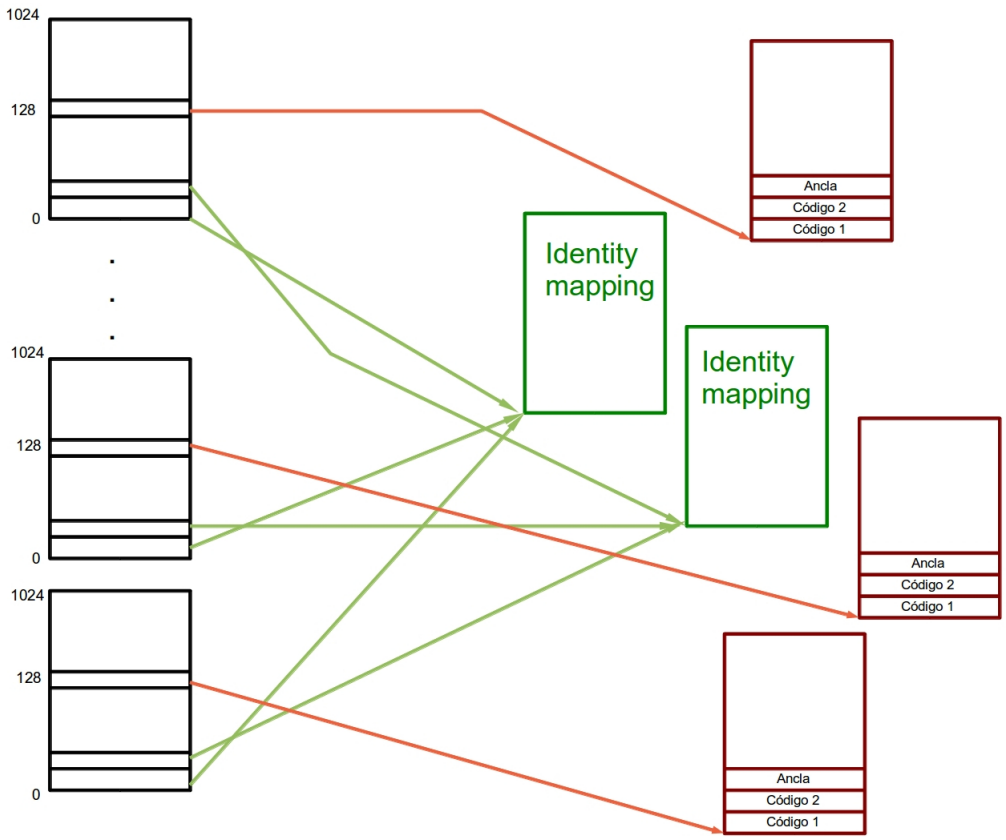

Figura 4: Esquema de las estructuras de paginación

Una vez que el sistema en ejecución, los mapeos de página se modifican en tiempo real mediante dos funciones que se encargan de crear mapeos de página y de borrar mapeos de página.

```
void mmu_mapear_pagina (unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned int attr)
void mmu_unmapear_pagina (unsigned int virtual, unsigned int cr3)
```

La función encargada de deshacer los mapeos termina con realizando un flush de la caché de paginación (la tlb) para asegurar la coherencia de todas las estructuras de paginación (al cambiar un mapeo puede pasar que el mapeo almacenado en la tlb no coincida más con la realidad).

Todo esto fue implementado en C. Tanto las funciones encargadas de crear las estructuras como las funciones encargadas de los mapeos. Lo único que se hace desde `kernel.asm` es llamar a esas funciones en el momento adecuado.

4.2. Activando la paginación

La paginación se activa mediante el bit de paginación, en el registro **cr0**. Es indispensable que, en el momento en que se activa ese bit, **cr3** ya esté seteado de manera adecuada y al menos las estructuras de paginación del kernel esten debidamenta cargadas en memoria.

5. Interrupciones

Las interrupciones y las excepciones son el método a través del cuál se logra interactuar con agentes externos, asegurar la integridad del sistema ante la ocurrencia de errores, o administrar los recursos del sistema. Para cada posible interrupción que se desea atender es necesario definir un handler de interrupción, o rutina de atención, y declararla a través de un descriptor en una estructura del sistema denominada Interrupt Descriptor Table (IDT).

En el sistema implementado se realizaron esencialmente los siguientes tipos de hanlders de interrupción: aquellos donde predomina la lógica con respecto al control de los recursos de sistema, como la interrupción generada por el reloj (que marca la unidad de tiempo sobre la que funciona el scheduler), la rutina de atención que procesa los eventos generados por el teclado, los handlers de atención a las syscalls (implementando servicios para las tareas del sistema), y las rutinas que proveen la lógica destinada a asegurar la integridad del sistema como puede ante el caso de una excepción como por ejemplo un Page Fault o un Segmentation Fault.

5.1. Interrupciones de control de flujo y administración de sistema

La principal encargada de controlar el flujo de ejecución en el sistema es la int0x32: la interrupción generada por el reloj. Esta interrupción funciona de manera muy cercana al scheduler, consultándole cuál tarea es la próxima a ejecutar y realizando el cambio de tarea de ser necesario. El scheduler implementado tiene un valor de retorno distinguido: si retorna un valor igual a 0, significa que la tarea ejecutándose actualmente debe seguir haciéndolo, si por el contrario devuelve cualquier otro número se lo interpreta como un índice en la GDT al que se debe realizar un `jmp far`, suscitando un cambio de contexto automático.

Por otra parte, antes de realizar el `jmp far` es importante verificar si la tarea a la que se va a saltar es una bandera o un navío. Antes de saltar a una bandera, se reinicializa su TSS para que el EIP vuelva a apuntar al comienzo de la función. Si esto no se hiciera, la ejecución seguiría a partir de la interrupción de fin de bandera y ese no es el compartamiento esperado. En los navíos, en cambio, sí se espera que la ejecución continúe donde fue interrumpida.

Es importante contemplar el hecho de que cuando se retoma la ejecución de una tarea, su EIP va a estar ubicado adentro de la interrupción donde se detuvo, por lo tanto tiene que siempre haber un JMP al final del llamado, incluso luego de un `jmp far`.

Por último todos los handlers referidos a las excepciones de sistema (page fault, etc) al final también cambian el flujo de ejecución pues luego de la lógica referida a la seguridad tienen que retomar el flujo de ejecución a algún lado, ya sea al lugar donde vino o , en caso de que esa tarea haya sido desalojada, a algún otro lado (la tarea idle).

5.2. Interrupciones de seguridad

Las excepciones de sistema como *General Protection* son el mejor ejemplo. El algoritmo de estas interrupciones es sencillo, lo que hace es verificar que quién produjo esa excepción sea efectivamente una tarea. Si fue una tarea la desaloja, sino Interrumpe el sistema y avisa que hubo una excepción. Luego de desalojar la tarea sencillamente salta hacia la tarea idle. Además todas estas interrupciones tienen exactamente la misma lógica, por lo que en principio se podrían implementar todas con un macro de nasm. El único código específico que implementamos para cada una es una funcion de C que actualiza la pantalla de manera adecuada. Sobre esto se hablará mas en detalla en la sección de pantalla.

La syscalls también tienen que hacer algunas verificaciones de seguridad. Las banderas no pueden llamar a las funciones de los barcos y los barcos no pueden llamar a la interrupción de final de bandera.

La interrupción de reloj debe verificar que el contexto en el que se está ejecutando no sea el de una bandera, pues si es así debe hacer desalojarla.

6. Manejo de tareas

El manejo de tareas tiene varias aristas. Por un lado está la gestión "física" de las tareas. Es decir como se cambia entre el contexto de una y el contexto de otra. Este punto se manejo mediante el mecanismo automático que provee la arquitectura intel.

Para esto hubo que crear tss para cada tarea. Además por cuestiones de facilitar la gestión la tareas se decidió crear una tss extra para cada bandera. De esta forma cada navio tiene 2 tss, una para correr su código de acción y otra para correr su código de bandera. Además hay 2 tss extra, una para la tarea idle y otra auxiliar (*tarea inicial*), para poder realizar el primer salto sin inconvenientes.

Para poder primer cambio de contexto (`JMP FAR`) sobre esta tss auxiliar previamente se setea el TR. Este seteo está hardcodeado porque el descriptor de la tss auxiliar siempre se guarda exactamente en el mismo índice de la gdt.

Todos los cambios de texto se hacen mediante un `JMP FAR`. En ningún momento se usa ningún otro mecanismo. El problema que surge con esto es que las banderas no tienen un código cíclico. Es decir, las banderas ejecutan una porción de código que termina, al terminar su contexto queda estático en la posición en la que terminó, por lo tanto si se llama a esa bandera otra vez en ese mismo contexto sin hacer nada la bandera va a ejecutar cosas indebidas. Para subsanar ese inconveniente siempre antes de saltar una bandera se vuelve a inicializar su tss, de esta forma nos aseguramos de que el contexto siempre sea el contexto inicial.

7. Protección

Las tareas corren en el anillo de seguridad 3. El kernel en anillo 0. A nivel paginación las tareas son usuario y el kernel es supervisor.

Las tareas no pueden acceder a memoria del kernel bajo ninguna circinstancia. Esto está asegurado en el mapa de memoria. El mismo si bien incluye las tablas de memoria del kernel las incluye con un privilegios mayor, por lo tanto si una tarea intenta acceder a esas páginas cae en un page fault que termina en el desalojo de esa tarea.

El kernel está programado para no tener que incurrir en ninguna excepción especial, por lo tanto el código que se ejecuta cuando ocurre una excepción es siempre el mismo. Desalojar a esa tarea y saltar a la tarea idle. Esto asegura que toda tarea que realice una acción ilegal que produzca uan excepción (dividir entre cero, seg fault, page fault, etc) será desalojada.

Además hay un par de cuestiones extra. Por ejemplo, una bandera no puede llamar a una int0x50. Si hace esto inmediatamente es desalojada.

8. Scheduler

Dadas las dieciséis tareas del sistema -ocho navíos, ocho banderas-, el scheduler desarrollado realiza un ciclo de ejecución sobre todas las tareas alternando de la siguiente forma: se dedican tres tics de reloj a ejecutar navíos, se procesan todas las funciones bandera pertenecientes a navíos activos, y luego se retoma la ejecución de los navíos. Los navíos se ejecutan secuencialmente por lo que todos reciben la misma cantidad de tics mientras no sean desalojados.

Para implementar esta lógica, se realizaron colas circulares basadas en arreglos para los índices de la tabla GDT de los navíos y las banderas[1]. Como no se ingresan nuevos elementos dinámicamente, el mecanismo de elminación de una tarea se puede realizar sencillamente marcando su entrada con un valor inválido, como el 0 ya que nunca es un índice válido en la tabla.

Para mantener el orden de ejecución descripto, se guardaron adicionalmente contadores para la cantidad de navíos activos en la cola, la cantidad de navíos pendientes hasta alternar con la ejecución de banderas, e inversamente la cantidad de banderas pendientes.

De esta forma, se mantiene el invariante de que al menos uno de los dos contadores de tareas pendientes es nulo, y se consumen elementos de la cola opuesta hasta terminar la ronda. Es decir, inicializando la cantidad de navíos pendientes a 3 y la cantidad de banderas pendientes a 0, se dedicarán 3 tics a ejecutar navíos, y luego se asigna la cantidad de tareas activas al contador de banderas pendientes, asegurando que se llamen todas las banderas disponibles.

Como ya se dijo, cuando una tarea necesita ser desalojada, su entrada correspondiente y la de su navío o bandera asociada se anulan, permitiendo evitar su ejecución simplmente salteando los ceros que se encuentren en la cola.

[1] De los descriptores de sus tablas TSS, para ser preciso.

9. Conclusiones

A lo largo de todo el trabajo se presentó la enorme cantidad de inconvenientes que conlleva trabajar sobre la nada. Para realizar el trabajo fue imprecindible entender lo que estaba sucediendo en el kernel a muy bajo nivel, pues constanemente surgen inconvenientes cuya explicación no suele ser evidente.

También una cosa que se presentó es que la dificultad de la implementación asciende al infinito si uno lo permite. En todo momento se presentan ecrucijadas sobre como realizar algo específico teniendo una libertad casi absoluta al tener permisos de kernel. En general en esos momentos intentamos optar por la implementación mas clara con una lógica mas sólida perdiendo en algunos casos algo de performance. Sin embargo lo que se ganó con eso fue un código mucho mas manejable.