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

La arquitectura intel posee varios modos de trabajo. A lo largo del tp sólo usamos 2 de ellos: El modo real y el modo protegido. Todo los procesadores con arquitectura intel x86 en el momento en que arrancan lo hacen en modo real. En modo real el procesador posee la misma interfaz que un intel 8086. Esto conlleva un tamaño de palabra de 16 bits, capacidad de dirección de 1mb, nula protección por hardware, etc, etc.

En modo protegido, por el contrario, el procesador puede hacer pleno uso de sus recursos. Una vez que se pasa a modo protegido el procesador puede hacer pleno uso de todo su set de instrucciones, direccionar a 4gb de RAM, utilizar potentes mecanismos de protección por hardware, etc.

2.1. Modo real

Trabajando en modo real uno está realmente limitado, por lo que hicimos todo lo posible por salir lo mas rápido posible de el. Sin embarg algunas cosas se hacen en modo real para ahorrar inconvenientes luego. La mayoría de estas cosas, sin embargo, fueron implementadas por la cátedra.

La primera, por supuesto, es correr el bootloader. El bootloader se ejecuta por completo en modo real. Una vez que el bootloader termina lo siguiente que se hace es deshabilitar las instrucciones. Muy importante pues todavía no se definió como manejarlas, por lo que si esto no se hace en cuanto cae la primera interrupción de reloj se genera una excepción de la que no se puede salir, pues todavía no se declaró en ninguna parte que hacer con esa excepción. Luego hay que habilitar A20. Esto es buena idea hacerlo antes de saltar a modo protegido, cuando de todas maneras el límite de direccionamiento es 1 mega. Lo último que se hace antes de pasar a modo protegido es habilitar los 16 colores de fondo de video. Por defecto el video está seteado para que el bit mas significativo de las casillas de formato represente "blink", es decir si el caracter parpadea o no. Lo que hicimos, entonces , fue setear el video para que ese bit se agruegue a la paleta de colores del fondo de pantalla. De esta manera se obtiene la posibilidad de crear una pantalla un poco mas expresiva. Esto decidimos hacerlo aún en modo real principalmente por el motivo de hacer todo lo relacionado a la comunicación con el hardware en la etapa mas temprana posible. Es decir .[en]listar.[el] sistema antes de correrlo.

Una vez terminadas estas tareas se pasa a modo protegido.

2.2. Pasaje a modo protegido

El pasaje a modo protegido es una parte bastante delicada. En primer lugar es importante armar una gdt aunque sea minimal y setear el gdtr de manera adecuada. Sobre eso vamos a entrar en mas detalle en la sección correspondiente.

Una vez que esto está asegurado se debe efectivamente pasar a modo protegido. Esto, en principio, no es mas que setear un bit en el registro de control CR0. Sin embargo en el momento en que se setea ese bit cambia la manera de interpretar a los registros de segmento, por lo que la siguiente instrucción es altamente probable que cause algún problema al respecto. Lo que se debe hacer entonces es setear los registros de segmento en valores empezando por el CS, que es el primero que puede producir algún tipo de problema. Para esto lo que se hace es un jmp far a la siguiente instrucción hardcodeando el valor adecuado del CS.

```
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

Ya en modo protegido es importante realizar 2 cosas antes que nada. Estas son acomodar el resto de los registros de segmento (el JMP FAR setea el CS, pero faltan todos los demás) y setear la pila como corresponde. Esto último consiste sencillamente en darle a ESP y a EBP los valores adecuados para que la pila quede donde nosotros queremos (en nuestro caso eso es la posición de memoria 0x27000 de manera tal que la pila viva entre 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. En este trabajo sólo utilizamos 3 tipos de descriptores: El primero y mas trivial es un descriptor nulo. El procesador produce una excepción automáticamente cunado se intenta acceder a la posición cero de la GDT. De esta manera el primer espacio de la tabla se rellena con un descriptor nulo para evitar confusiones. Es decir que este descriptor en realidad cumple una función accesoria, no modifica en nada el funcionamiento del sistema. Sin embargo es muy cómodo que este ahí. El segundo tipo de descriptor le corresponde a los descriptores de segmento. Los mismos delimentan el tamaño y la ubicación de los segmentos, así como sus propiedades (quién los puede acceder, que clase código hay adentro, etc, etc. Por último se utilizaron descriptores de TSS para realizar el salto automático de tareas (sobre esto vamos a hablar mas adelante con mayor profundidad, ahora sólo se explicará la parte que concierne específicamente a la GDT).

3.1. Segmentación flat

Como indica el enunciado todo el trabajo funciona con segmentación flat. Esto significa dejar de lado la protección por segmentación armando 4 segmentos que ocupen toda la memoria superpuestos entre si. Estos 4 segmentos son 2 con nivel de privilegio cero (máximo nivel de provilegios) y 2 con nivel de privilegio 3 (mínimo nivel de privilegios), y para cada nivel un segmento de código y uno de datos.

Por supuesto que esta práctica acarrea problemas, algunos de ellos graves como por ejemplo que se pierde toda la protección por hardware brindada por la segmentación. La contracara de esto es que se obtiene un entorno mucho mas amistoso para programar, y la mayoría de la seguridad que se pierde por la segmentación se puede recuperar utilizando de manera adecuado la paginación.

3.2. Descriptores de segmento

Los descriptores de segmento se hardcodean en tiempo de compilación. Es decir que en el mismo código la GDT ya contiene sus descriptores de segmento con los parámetros adecuados. Es importante que esto se puede realizar exclusivamente porque todos lo necesario para completar esos descriptores se conoce de antemano. Esto en parte es una consecuencia de la segmentación flat.

Una vez que el programa empieza a correr estos descriptores de segmento no se vuelven a modificar. Se trabaja siempre los segmentos flat, por lo que no hace falta modificar ni agregar nada. Quedan estáticos para siempre.

Además de los segmentos flat se crea un segmento que contiene la memoria de video. Este segmento se utilizó con fines didácticos en un principio, pero luego todas las funciones de video que se utilizan a lo largo del trabajo acceden a la memoria de video por medio del segmento flat de datos de nivel 0.

3.2.1. Atributos descriptores de segmento

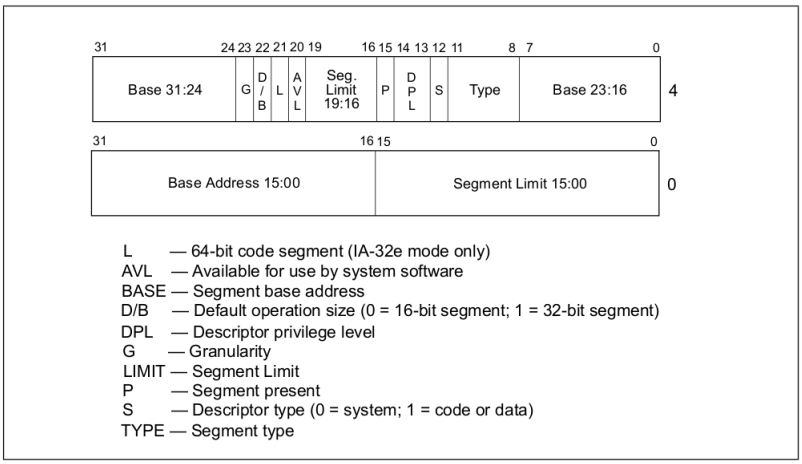

Figura 1: Descriptor de segmento

3.3. Descriptores de TSS

Para los descriptores de TSS se eligió otra dinámica. Las TSS se encuentran ubicadas en un arreglo de TSS que se crea dinámicamente en tiempo de ejecución, por lo que en el momento de la compilación no se sabe el lugar donde va a estar y por lo tanto no se puede hardcodear esa dirección.

Lo que se hizo, entonces, fue agregar los descriptores de TSSde manera dinámica en tiempo de ejecución. Para esto se crearon 4 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.

La siguiente gran ventaja de todo esto 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 va a hablar sobre la seguridad de manera específica es importante mencionar que debido a que la segmentación flat detruye todos los sistemas de seguridad que provee la segmentación la paginación se debe tratar con mucho cuidado, pues este mecanismo tiene que asegurar toda la seguridad y la integridad de la memoria.

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. Pero además existen tablas de páginas compartidas y tablas de páginas privadas. Además 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 inmplementó tal cuál lo dice el enunciado. Hay identity mapping en los primeros 8mb y después cada tarea tiene los mapeos que necesita para acceder a su código y al ancla.

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 cada tarea, 1 para el kernel, 1 page table privado de cada tarea (con permisos de usuario) y 2 page table compartidos(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 de las que se hablará mas adelante.

La otra acción que se realiza en este momento es la de copiar el código de las tareas de la tierra al mar. Elegimos hacerlo en este momento porque así ya se pueden hacer los mapeos y dejar todo todo listo para el momento en que empicen a correr las tareas.

Todo explicado recién se hacer mediante una sola función escrita en c que luego se llama desde kernel. asm

Los page table compartidos tienen los mapeos de las direcciones del kernel. Esas direcciones estar mapeadas en todas las tareas y con los mismos atributos, motivo por le cuál decidimos hacerlos comunes. Esto significa todas las primeras entradas de page dir apuntan al mismo page table, el cuál redirecciona con permisos de administrador a los primeros 4mb de kernel, del mismo modo todas las segundas entradas de page dir apuntan también a una page table común que direcciona a la segunda parte del kernel.

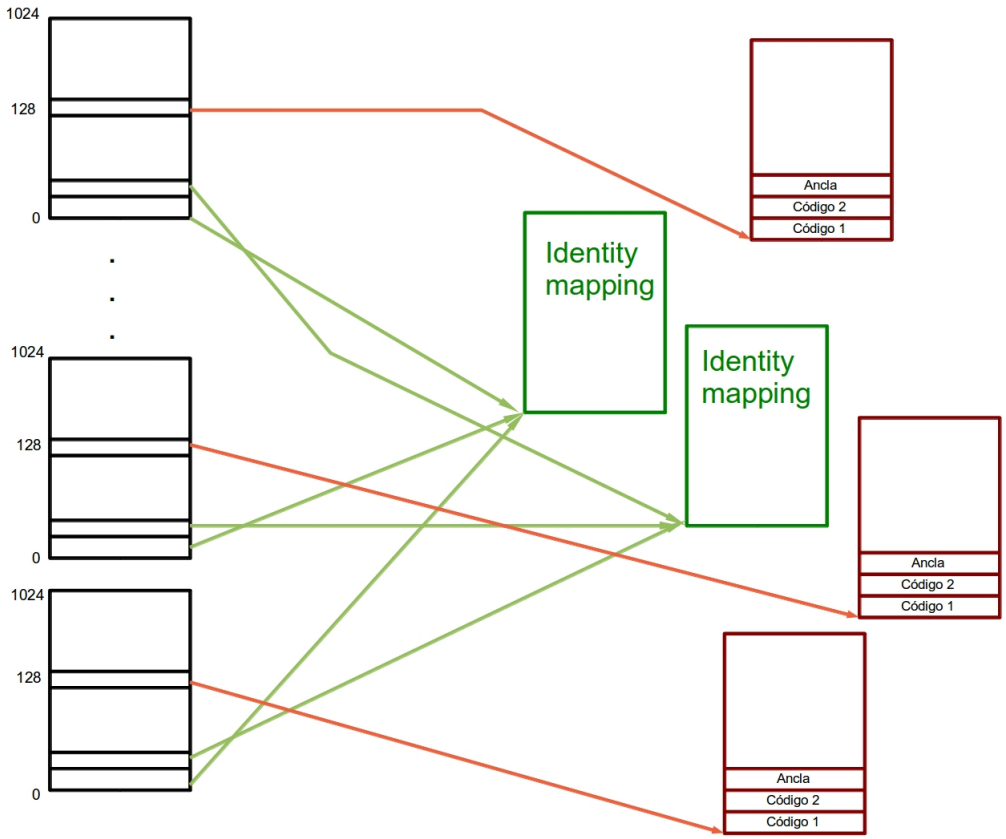

Figura 2: Esquema de las estructuras de paginación

Una vez que el sistema está corriendo 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 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 activa 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 administrar los recursos del sistema y asegurar la integridad del sistema. Todos los handlers de interrupción fueron pensados y escritos con esos objetivos en mente.

De esta manera en cada handler se pueden notar esas dos aristas, en algunos handlers predomina la lógica con respecto al control de los recursos de sistema, como en la interrupción de reloj que se maneja codo a codo con el scheduller administrar el tiempo de ejecución entre las distintas tareas. En otras, por el contrario, lo que mas se aprecia es la lógica destinada a asegurar la integridad del sistema como puede ser el caso de una excepción generada por un page fault o un seg fault.

5.1. Interrupciones de control de flujo y administración de sistema

La principal encarga de esto es la int0x32, es decir la interrupción que produce el reloj. Esta interrupción funcina de manera muy cercana al escheduller y su función basicamente es saltar a la tarea a la que le toca ejecutar o bien, no hacer nada en caso de que la tarea que está corriendo deba continuar. Para tomar esta desición lo que hace es consultarle al scheduller que tarea debe seguir corriendo. Si el escheduller devuelve 0 significa que la tarea que corre 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 para realizar 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 re inicializamos su TSS para que el EIP vuelva a apuntar a donde es debido, 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, por otra parte , no hace falta hacer nada en espececial, basta realizar el salto para que las cosas funcionen.

Es importante que en el código de la interrupción se contempla que cuando a la tarea le vuelva a tocar su eip va a estar ubicado adentro de la interrupción, por lo tanto tiene que siempre haber un JMP al final de la interrupción, incluso luego de un JMP FAR.

Otra interrupción que controla importantemente el control de flujo es la interrupción de teclado. De hecho la misma no tiene ningún control de seguridad. Sencillamente es un gran switch donde se verifica que tecla se tocó y en caso de caer en alguna de las teclas contempladas (“e”,“m”,“0-9”) entonces se realiza el efecto esperado. Las teclas “e” y “m” se manejan mediante funciones escritas en C.

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, 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 siemre 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 uqe terminó, por lo tanto se 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 tablasde memoria del kernel las incluye con un privilegios mayor, por lo tanto si una tarea intenta acceder a esas páginas cae un page fault que termina en el desalojo de esa tarea.

El kernel está programado para no tener que incurrir en ninguna excepción especial, lo tanto el código que se ejecuta cuando ocurre una excepción es siempre el mismo. Desalojar a esa tarea y saltar a la tarea iddle. 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.