Trabajo Práctico 3
System Programming - `Batalla Bytal`

Organización del Computador 2

Segundo Cuatrimestre 2016

1. Objetivo

El tercer trabajo práctico de la materia consiste en un conjunto de ejercicios en los que se aplican de forma gradual, los conceptos de *System Programming* vistos en las clases teóricas y prácticas.

Se busca construir un sistema mínimo que permita correr exactamente 8 tareas concurrentemente a nivel de usuario. El sistema será capaz de capturar cualquier problema que puedan generar las tareas y tomar las acciones necesarias para quitar a la tarea del sistema.

Los ejercicios de este trabajo práctico proponen utilizar los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo enfocados en dos aspectos: el sistema de protección y la ejecución concurrente de tareas.

2. Introducción

Para este trabajo se utilizará como entorno de pruebas el programa *Bochs*. El mismo permite simular una computadora IBM-PC compatible desde el inicio, y realizar tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en *Bochs* de forma sencilla.

Una computadora al iniciar comienza con la ejecución del POST y el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En este caso dispondremos de un *Floppy Disk* como dispositivo de booteo. En el primer sector de dicho *floppy*, se almacena el *boot-sector*. El BIOS se encarga de copiar a memoria 512 bytes del sector, a partir de la dirección `0x7c00`. Luego, se comienza a ejecutar el código a partir esta dirección. El boot-sector debe encontrar en el *floppy* el archivo `kernel.bin` y copiarlo a memoria. Éste se copia a partir de la dirección `0x1200`, y luego se ejecuta a partir de esa misma dirección. En la figura 1 se presenta el mapa de organización de la memoria utilizada por el *kernel*.

Es importante tener en cuenta que el código del *boot-sector* se encarga exclusivamente de copiar el *kernel* y dar el control al mismo, es decir, no cambia el modo del procesador. El código del *boot-sector*, como así todo el esquema de trabajo para armar el *kernel* y correr tareas, es provisto por la cátedra.

Los archivos a utilizar como punto de partida para este trabajo práctico son los siguientes:

- `Makefile` - encargado de compilar y generar el *floppy disk*.
- `bochsrc` y `bochsdbg` - configuración para inicializar Bochs.
- `diskette.img` - la imagen del *floppy* que contiene el *boot-sector* preparado para cargar el *kernel*. (viene comprimida, la deben descomprimir)
- `kernel.asm` - esquema básico del código para el *kernel*.
- `defines.h` y `colors.h` - constantes y definiciones
- `gdt.h` y `gdt.c` - definición de la tabla de descriptores globales.
- `tss.h` y `tss.c` - definición de entradas de TSS.
- `idt.h` y `idt.c` - entradas para la IDT y funciones asociadas como `idt_inicializar` para completar entradas en la IDT.
- `isr.h` y `isr.asm` - definiciones de las rutinas para atender interrupciones (*Interrupt Service Routines*) y la definición de la función `screen_proximo_reloj`.
- `sched.h` y `sched.c` - rutinas asociadas al *scheduler*.
- `mmu.h` y `mmu.c` - rutinas asociadas a la administración de memoria.
- `screen.h` y `screen.c` - rutinas para pintar la pantalla.
- `a20.asm` - rutinas para habilitar y deshabilitar A20.
- `imprimir.mac` - macros útiles para imprimir por pantalla y transformar valores.
- `idle.asm` - código de la tarea `Idle`.
- `game.h` y `game.c` - implementación de los llamados al sistema.
- `syscalls.h` - interfaz utilizar en C los llamados al sistema.
- `genbin.c` - programa para generar el binario de las tareas.
- `tarea1.c` a `tarea8.c` - código de dummy de las tareas.
- `i386.h` - funciones auxiliares para utilizar *assembly* desde C.
- `pic.c` y `pic.h` - funciones `habilitar_pic`, `deshabilitar_pic`, `fin_intr_pic1` y `resetear_pic`.

Todos los archivos provistos por la cátedra pueden y deben ser modificados. Los mismos sirven como guía del trabajo y están armados de esa forma, es decir, que antes de utilizar cualquier parte del código deben entenderla y modificarla para que cumpla con las especificaciones de su propio sistema.

A continuación se da paso al enunciado, se recomienda leerlo en su totalidad antes de comenzar con los ejercicios. El núcleo de los ejercicios será realizado en clase, dejando cuestiones cosméticas y de informe para el hogar.

3. Batalla Bytal

En 1930 varias compañías entre ellas Milton Bradley Company, comenzaron a comercializar el juego hoy conocido como *Battleship*, en español "Batalla Naval" (o "Hundir la Flota" como fue conocido en España). Las primeras versiones del juego estaban basadas en lápiz y papel; no fue hasta 1967 que se introdujo la versión más conocida en piezas y tablero de plástico. Desde entonces se ha convertido en un clásico en el que participan dos jugadores y consiste en adivinar las posiciones de los barcos del contrincante.

En este trabajo práctico vamos a implementar una adaptación del juego que consiste en "hundir" tareas cambiando su código.

El mar de bytes sobre el que navegarán las tareas lo denominaremos sencillamente *mar* y será un área de memoria con nivel de protección de usuario. Por otro lado, tendremos en nuestro sistema un área de memoria denominada *tierra* que contendrá todas las estructuras y código del *kernel*.

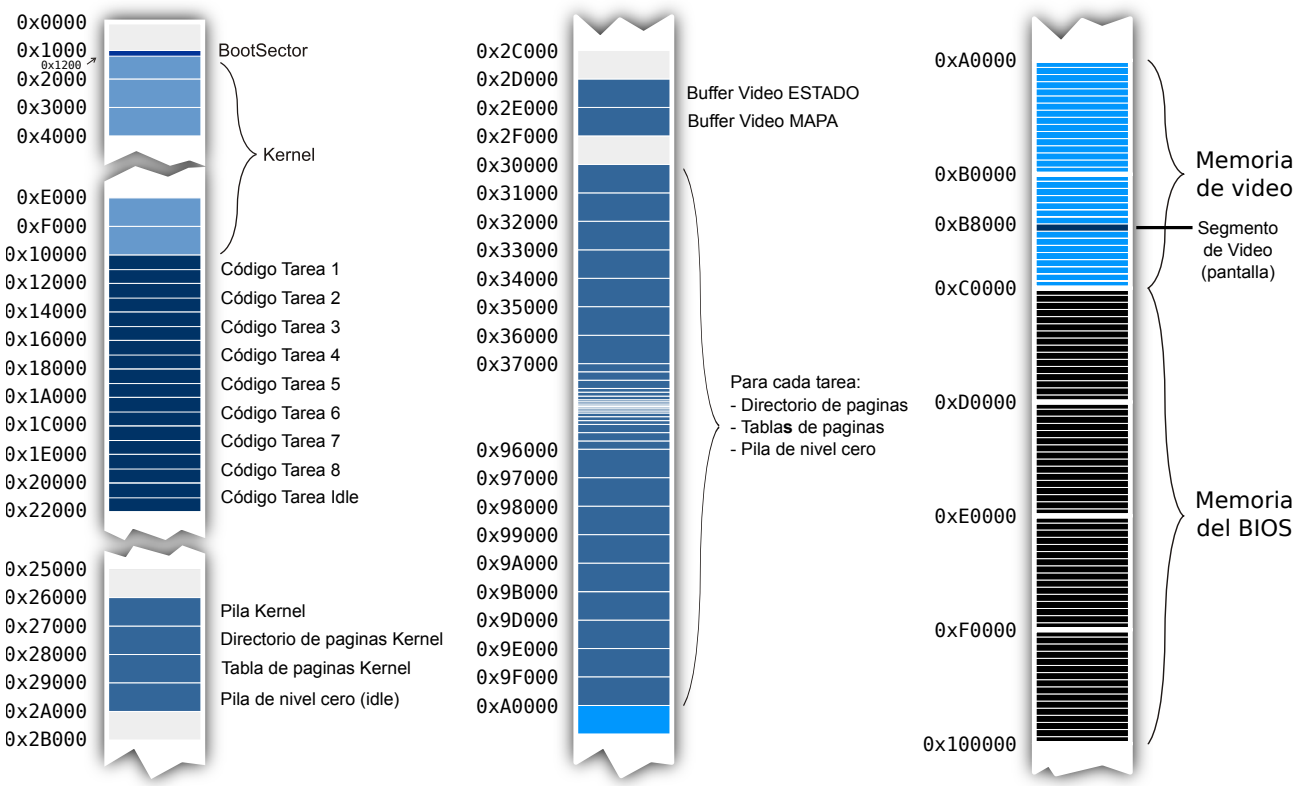

Figura 1: Mapa de la organización de la memoria física del *kernel*.

La totalidad de la memoria física a utilizar va a estar dividida en estas dos áreas mencionadas anteriormente *mar*(usuario) y *tierra*(*kernel*). El espacio de tierra ocupará exactamente el primer megabyte de memoria, mientras que el mar ocupará los siguientes 6.5MB.

3.1. Cruceros de Bytes (Tareas)

El sistema correrá 8 tareas o Cruceros concurrentemente. Los Cruceros podrán realizar tres acciones, "tirar un ancla a tierra", "lanzar misiles" o "navegar". Estas acciones serán descriptas más adelante como servicios que provee el sistema para con las tareas.

El mapa de memoria de la tarea estará compuesto únicamente por dos páginas de memoria, es decir el espacio de usuario asignado para cada tarea será de 8kb. Además las tareas tienen mapeadas una página dentro del espacio de memoria del *kernel* (tierra), esta página está mapeada como solo lectura. Este mapeo en solo lectura lo llamaremos "ancla".

Figura 2: Ejemplo de ancla.

3.1.1. Reglas de navegación (Servicios del Sistema)

Las tareas pueden solicitar tres servicios al sistema, estos servicios deben ser implementados como parte del trabajo práctico. Los servicios son llamados al sistema (*system calls*) que se realizan mediante la interrupción `0x50`. El registro `EAX` indica cual de los servicios se quiere solicitar. A continuación se describe el funcionamiento de cada uno de los servicios. En la figura 3 se ilustra el comportamiento de los tres llamados al sistema.

- **"tirar un ancla a tierra" (leer memoria del *kernel*)**

  Como fue mencionado anteriormente, las tareas tienen mapeada una página de memoria del *kernel* como solo lectura ("ancla"). Esta página puede ser remapeada a cualquier posición de memoria dentro del espacio de *kernel* (tierra). Para realizar este remapeo se utiliza el servicio `fondear` que equivale a "tirar una ancla a tierra"(ver figura 2), es decir, remapear la página de *kernel* a una dirección física diferente.

  → `fondear`

  `EAX = 0x923`

  `EBX =` dirección física del área de tierra a mapear (múltiplo de 4 k)

- **"lanzar misiles" (escribir memoria de usuario)**

  Los valientes corsarios que navegan estos intrépidos navíos deben tener alguna herramienta para defenderse de las malignas amenazas que aparecen en los tumultuosos mares de bytes. Para esto se cuenta con un servicio denominado "cañonear" que permite lanzar un misil de bytes a cualquier posición del mar. Este servicio permite escribir en cualquier posición de memoria del área de usuario la exacta cantidad de 97 bytes.

  → `cañonear`

  `EAX = 0x83A`

  `EBX =` dirección física del área de usuario donde se disparará el misil.

  `ECX =` dirección relativa al espacio de la tarea donde se encuentra el buffer de 97 bytes que contiene al misil.

- **"navegar" (modificar el mapeo de paginas de la tarea)**

  Las tareas son temerarios navíos que navegan los mares de bytes, para esto se cuenta con el servicio "navegar". Este servicio permite copiar las páginas de código de una tarea a dos páginas cualquiera dentro del espacio de usuario.

  → `navegar`

  `EAX = 0xAEF`

  `EBX =` dirección física del área de usuario para la primer pagina de código

  `ECX =` dirección física del área de usuario para la segunda pagina de código

Una vez llamado cualquiera de estos servicios, el *scheduler* se encargará de desalojar a la tarea que lo llamo para dar paso a la próxima tarea. Este mecanismo será detallado mas adelante.

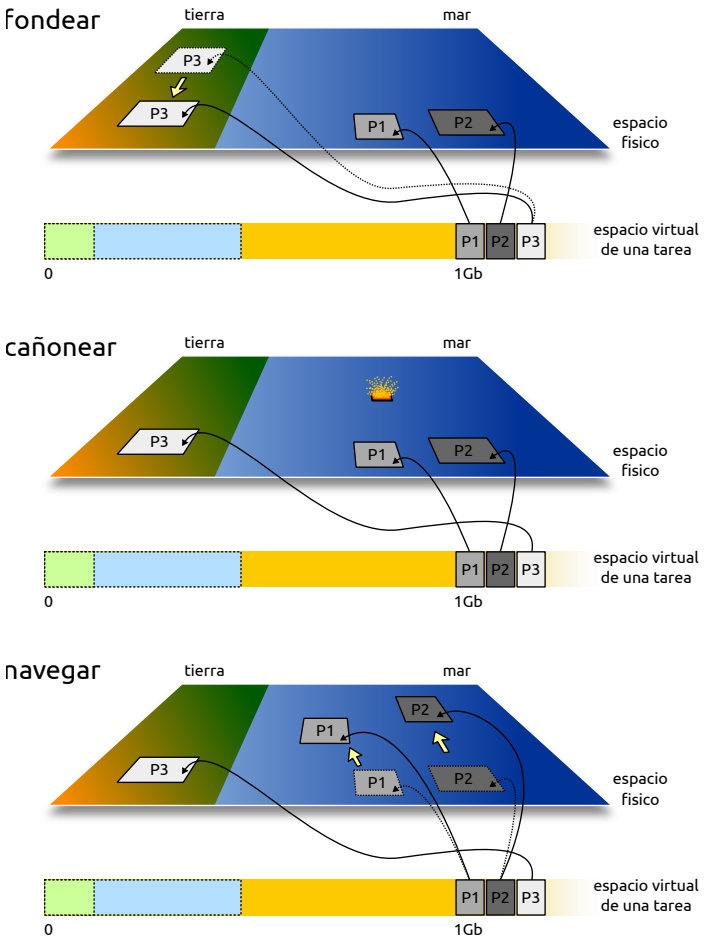

Figura 3: Ejemplo de Syscall

3.1.2. Jurisdicción marítima (Organización de la memoria de una tarea)

Cada una de las tareas tiene mapeadas las áreas del mar y la tierra con *identity mapping* en nivel 0, esto permite al *kernel* escribir en cualquier posición de memoria útil desde el contexto de cualquier tarea. Además las tareas mapean tres páginas para datos y código en nivel 3. Dos de estas páginas corresponderán a código y datos para la tarea, por lo que podrán ser escritas. La tercera página corresponderá al ancla y será de solo lectura. En la figura 4 se puede ver la posición de las páginas y donde deberán estar mapeadas como direcciones virtuales.

3.1.3. Bandera de identificación (Pseudo-Signal)

Todo navegante del mar sabe que cada barco navega con una bandera, nuestras tareas no son la excepción. Para esto cada tarea debe implementar una función que genere una bandera. Llamaremos bandera a un buffer de caracteres [1] de 10 columnas $\times$ 5 filas. En la vida real, las banderas suelen ser de tela, por lo que el viento en el mar indómito las hace

[1] Caracteres en pantalla, 1 byte de caracter y 1 byte de formato

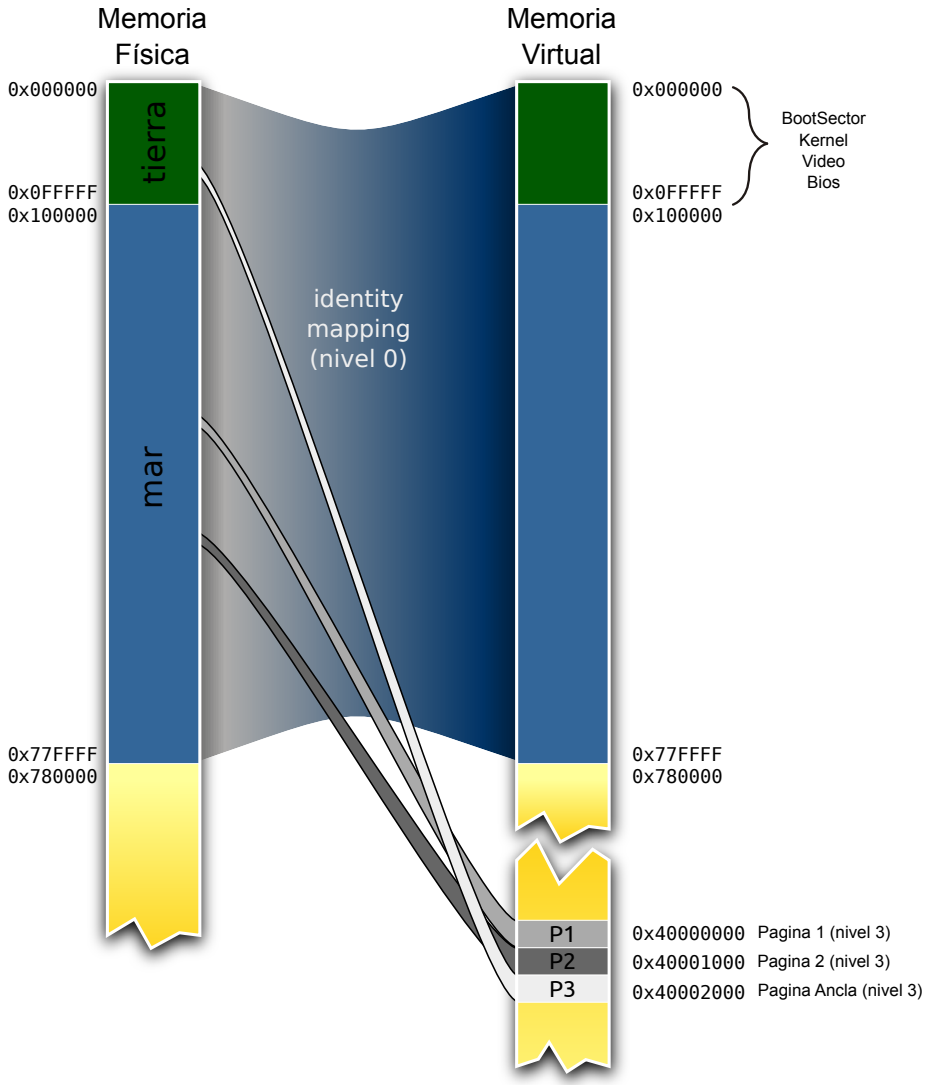

Figura 4: Mapa de memoria de la tarea

flamear holgadamente. Nuevamente, nuestras banderas no serán la excepción. La función que implementará cada tarea deberá generar un *buffer* distinto cada vez que se la llama, generando la ilusión de movimiento.

Por cada ciclo de reloj el sistema se encargará de llamar a la función "bandera" de cada una de las tareas, imprimiendo en pantalla el resultado. Para hacer posible este mecanismo, la tarea deberá respetar una estructura especial. En los últimos 4 bytes de los 8 kb que ocupa la tarea, se encontrará el puntero a la función "bandera". En la figura 5 se muestra la estructura que deberá respetar cada tarea.

La aridad de la función es: `unsigned char * bandera()`.

La implementación de la función "bandera" tiene algunas limitaciones. La función será ejecutada por el *kernel* en el contexto de la aplicación, por lo que contará con tan solo un *tick*

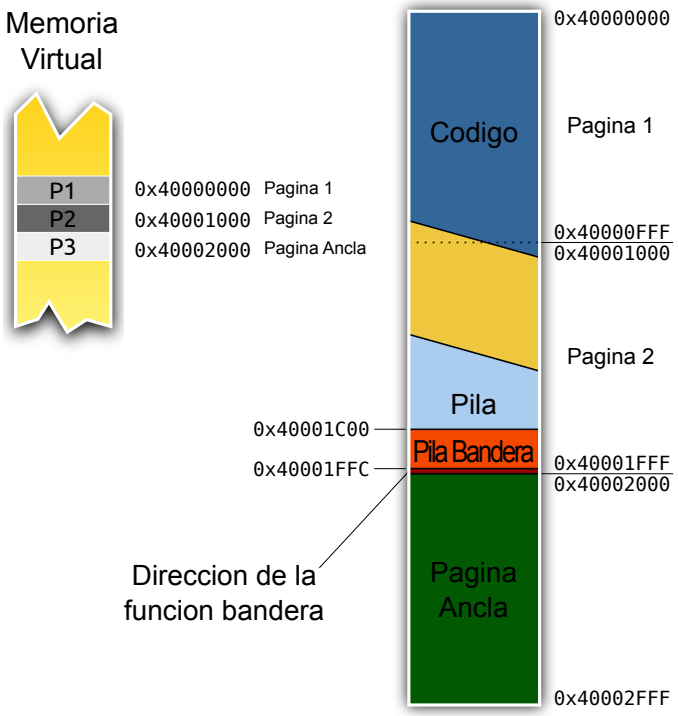

Figura 5: Mapa de memoria virtual de una tarea

de reloj para la ejecución completa de un llamado a la función. Si este tiempo es superado, entonces la tarea será **eliminada** del sistema. Para finalizar correctamente la ejecución de la bandera, se deberá llamar a la interrupción `0x66`, interrupción que será detallada más adelante. Esta interrupción será la encargada de capturar el resultado de la función "bandera" e imprimirlo en pantalla. Como la "bandera" será ejecutada como una tarea en nivel cero, se debe tener en cuenta el reservar espacio de *kernel* para la pila de nivel 0.

En la figura 6 se muestra como deberá ser escrito el *buffer*.

Figura 6: Buffer bandera

El sistema se encargará de llamar a la función "bandera" de cada una de las tareas cada 3 ciclos de reloj. Esto quiere decir que llegado ese número, el *scheduler* se encargará de modificar el contexto de ejecución de todas las tareas para hacer posible la ejecución de la función "bandera". Mas adelante en la sección 3.2.1 será explicado en detalle este procedimiento.

3.2. Scheduler

El sistema va a correr tareas de forma concurrente; una a una van a ser asignadas al procesador durante un tiempo. Para esto se va a contar con un *scheduler* minimal que se va a encargar de desalojar una tarea del procesador para intercambiarla por otra en intervalos regulares de tiempo.

Dado que las tareas pueden generar cualquier tipo de problema, se debe contar con un mecanismo que permita desalojarlas para que no puedan correr nunca más. Este mecanismo debe poder ser utilizado en cualquier contexto (durante una excepción, un acceso inválido a memoria, un error de protección general, etc.) o durante una interrupción para acceder a uno de los tres posibles servicios del sistema.

Cualquier acción que realice una tarea de forma incorrecta será penada con el desalojo de dicha tarea del sistema, es decir el hundimiento del buque.

Un punto fundamental en el diseño del *scheduler* es que debe proveer una funcionalidad para intercambiar cualquier tarea por la tarea `Idle`. Este mecanismo será utilizado al momento de llamar a cualquier servicio del sistema, ya que la tarea `Idle` será la encargada de completar el *quantum* de la tarea que llamó al servicio. La tarea `Idle` se ejecutará por el resto del *quantum* de la tarea desalojada, hasta que nuevamente se realice un intercambio de tareas por la próxima en la lista.

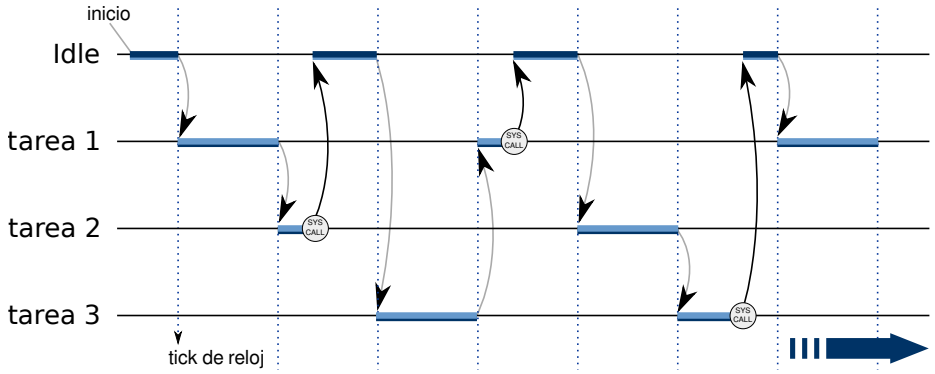

Figura 7: Ejemplo de funcionamiento del *Scheduler*

Inicialmente, la primera tarea en correr es la `Idle` como se puede ver en la figura 7. Luego, en el primer *tick* de reloj se lanza la tarea 1. Ésta corre hasta que termine su tiempo en el próximo *tick* de reloj o la tarea intente llamar a un servicio del sistema; de ser así, será desalojada y el tiempo restante será asignado a la tarea `Idle`. En la figura 7 esto sucede con la tarea 2.

Otra cuestión importante para el funcionamiento del *scheduler* es el mecanismo de "bandera". Este mecanismo es ejecutado por el *scheduler* cada 3 ciclos de reloj. Para esto se llamará a cada una de las funciones "bandera" de cada una de las tareas durante un ciclo de reloj (completando el tiempo restante del *quantum* con la tarea `Idle`).

3.2.1. Int `0x66`

El *scheduler* debe ser capaz de soportar el mecanismo de "bandera" explicado anteriormente. Para esto se provee una interrupción que SOLO puede ser llamada desde el contexto

de ejecución de la función “bandera” y permite a la tarea retornar a su contexto normal de ejecución. Si se da el caso en que una tarea llama a la interrupción `0x66` entonces esta tarea debe ser eliminada del sistema (lo que implica también eliminar a la bandera). En la figura 8 se ejemplifica el funcionamiento de la interrupción junto con el resto de las tareas e interrupción de reloj.

En el ejemplo de la figura se cuenta con tan solo 2 tareas y se realizarán los llamados a las funciones “bandera” cada tres ciclos de reloj. A continuación se detalla paso a paso la figura:

a - Se ejecuta la tarea 1 hasta la interrupción de reloj. La interrupción de reloj detecta que se ejecutaron tareas por tres ciclos, entonces ejecutará las funciones “bandera”.

b - Se cambia de contexto y se pasa a ejecutar la función “bandera” de la tarea 1. Esta se ejecuta hasta que llama a la interrupción `0x66` para salir de la función.

c - En la interrupción `0x66` se lanza la tarea `Idle` para terminar con el ciclo de reloj.

d - La tarea `Idle` corre hasta que termine el ciclo de reloj.

e - Se ejecuta la próxima función “bandera” utilizando el mismo mecanismo anterior

f - Una vez terminadas de ejecutar todas las funciones “bandera”, se pasa a ejecutar la próxima tarea del *scheduler*.

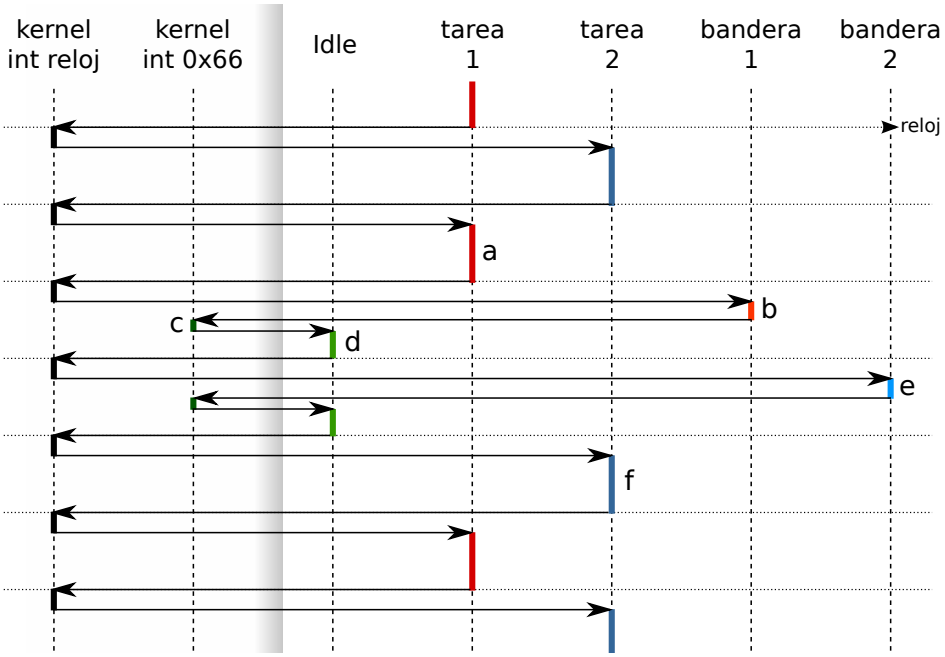

Figura 8: Ejemplo de la interrupción `0x66`

Si durante la ejecución de la función “bandera” se genera una interrupción de reloj, entonces la tarea correspondiente a esa bandera debe ser eliminada. Como fue mencionado anteriormente, la eliminación de una tarea del sistema corresponde a no volver a ejecutar nunca más ni el código de la tarea, ni su correspondiente bandera.

3.3. Pantalla

La pantalla estará controlada por el teclado, mediante la tecla “m” se activará el modo *mapa*, y mediante la tecla “e” se activará el modo *estado*.

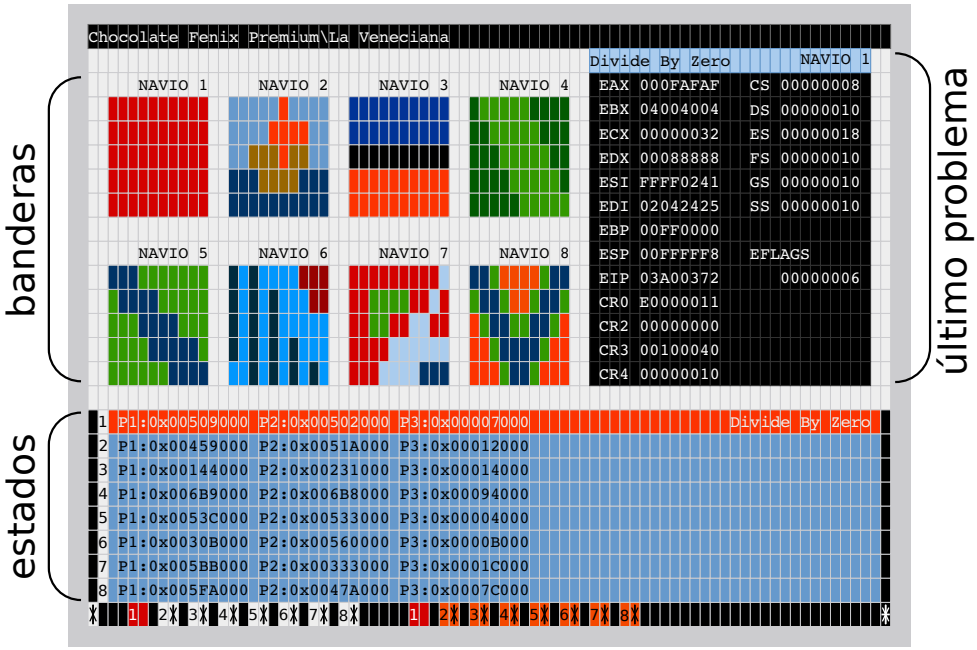

Figura 9: Pantalla en modo *estado*

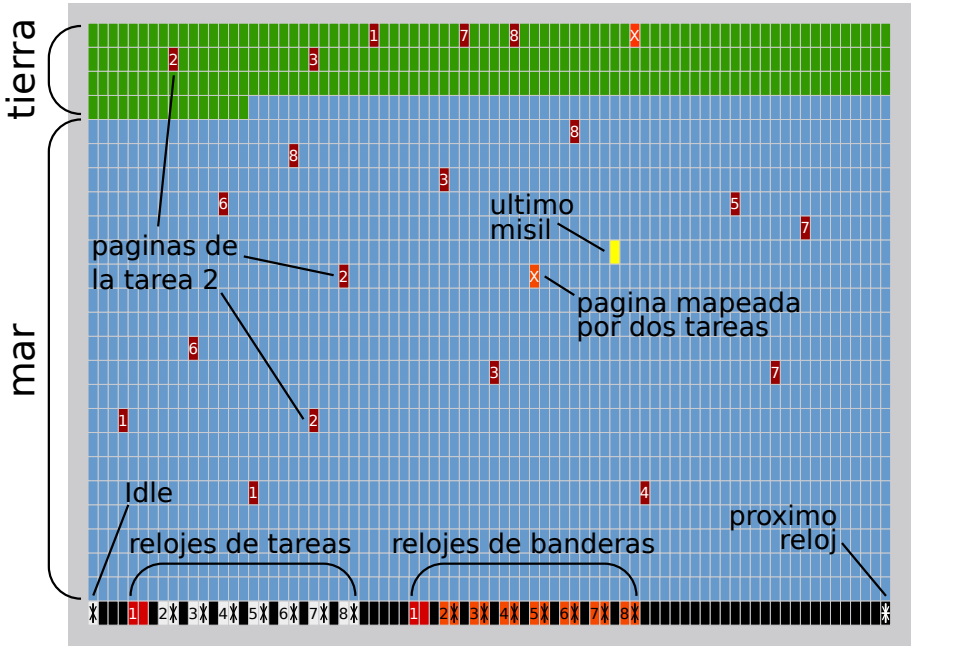

Figura 10: Pantalla en modo *mapa*

El modo *mapa* presentará un mapa en pantalla de toda la memoria accesible. Considerando que la pantalla tiene 80×25 caracteres y que la última línea se utilizará para información del sistema, restan 80×24 caracteres. Si cada caracter corresponde a una página de memoria, entonces podemos mostrar en el mapa 1920 páginas de memoria de 4 kb cada una. En resumen, nuestro mapa mostrará el total del mar y el kernel, que es de 7.5 MB; 1 MB correspondiente a *kernel* (tierra) y 6.5 MB correspondientes al espacio de usuario (mar).

La figura 10 y figura 9 muestran imágenes de cada pantalla indicando qué datos deben presentarse de forma mínima.

En la implementación las dos pantallas estarán guardadas en páginas del *kernel* que serán actualizadas cada vez que se produzca algún cambio. Se recomienda implementar funciones auxiliares que permitan imprimir datos en pantalla de forma cómoda, intercambiar pantallas e imprimir los *buffers* de las pantallas en la memoria de vídeo.

4. Ejercicios

4.1. Ejercicio 1

a) Completar la Tabla de Descriptores Globales (`GDT`) con 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 1.75GB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 17 posiciones se consideran utilizadas, por lo que el primer índice que deben usar para declarar los segmentos, es el 18 (contando desde cero).

b) Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección `0x27000`.

c) Declarar un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*.

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar la primer y última línea de color de fondo negro y letras blancas. Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior (para los próximos ejercicios se accederá a la memoria de vídeo por medio del segmento de datos de 1.75GB).

Nota: La `GDT` es un arreglo de `gdt_entry` declarado sólo una vez como `gdt`. El descriptor de la `GDT` en el código se llama `GDT_DESC`.

4.2. Ejercicio 2

a) Completar las entradas necesarias en la `IDT` para asociar diferentes rutinas a todas las excepciones del procesador. Cada rutina de excepción debe indicar en pantalla qué problema se produjo e interrumpir la ejecución. Posteriormente se modificarán estas rutinas para que se continúe la ejecución, resolviendo el problema y desalojando a la tarea que lo produjo.

b) Hacer lo necesario para que el procesador utilice la IDT creada anteriormente. Generar una excepción para probarla.

Nota: La `IDT` es un arreglo de `idt_entry` declarado solo una vez como `idt`. El descriptor de la `IDT` en el código se llama `IDT_DESC`. Para inicializar la `IDT` se debe invocar la función `idt_inicializar`.

4.3. Ejercicio 3

a) Escribir una rutina que se encargue de limpiar el *buffer* de vídeo y pintarlo como indican las figuras 9 y 10. Tener en cuenta que deben ser escritos de forma genérica para posteriormente ser completados con información del sistema. Además considerar estas imágenes como sugerencias, ya que pueden ser modificadas a gusto según cada grupo mostrando siempre la misma información.

b) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el *kernel* (`mmu_inicializar_dir_kernel`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x0077FFFF`, como ilustra la figura

4. Además, esta función debe inicializar el directorio de páginas en la dirección `0x27000` y la tabla de páginas en `0x28000` (ver fig. 1).

c) Completar el código necesario para activar paginación.

d) Escribir una rutina que imprima el nombre del grupo en pantalla. Debe estar ubicado en la primer linea de la pantalla a partir del segundo caracter.

4.4. Ejercicio 4

a) Escribir una rutina (`mmu_inicializar_dir_tarea`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 4. La rutina debe copiar el código de la tarea a su área asignada, es decir sus dos páginas de código dentro del *mar* y mapear dichas páginas a partir de la dirección virtual `0x40000000`(1GB). Además debe mapear la página del ancla a la primer pagina de memoria (`0x0`).

b) Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.

I- `mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica)`
Permite mapear la página física correspondiente a `fisica` en la dirección virtual `virtual` utilizando `cr3`.

II- `mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

c) Construir un mapa de memoria para tareas e intercambiarlo con el del *kernel*, luego cambiar el color del fondo del primer caracter de la pantalla y volver a la normalidad.

Nota: Por la construcción del *kernel*, las direcciones de los los mapas de memoria (`page directory` y `page table`) están mapeadas con *identity mapping*. En los ejercicios en donde se modifica el directorio o tabla de páginas, hay que llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

4.5. Ejercicio 5

a) Completar las entradas necesarias en la `IDT` para asociar una rutina a la interrupción del reloj, otra a la interrupción de teclado y por último a dos interrupciones de software `0x50` y `0x66`.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `screen_proximo_reloj`. La misma se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla. La función `proximo_reloj` está definida en `isr.asm`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquier número, se presente el mismo en la esquina superior derecha de la pantalla. El número debe ser escrito en color blanco con fondo de color aleatorio por cada tecla que sea presionada [2]. Además se deben mostrar los *buffers* de vídeo según indica el enunciado al presionar las teclas “m” y “e”.

[2] http://wiki.osdev.org/Text_UI

d) Escribir la rutina asociada a la interrupción `0x50` y `0x66` para que modifique el valor de `eax` por $0x42$. Posteriormente este comportamiento va a ser modificado para atender los servicios del sistema.

4.6. Ejercicio 6

a) Definir 18 entradas en la `GDT`. Una reservada para la `tarea_inicial`, otra para la tarea `Idle` y las 16 restantes, dos para cada tarea que van a ejecutar el sistema.

b) Completar la entrada de la `TSS` correspondiente a la tarea `Idle`. Esta información se encuentra en el archivo `TSS.C`. La tarea `Idle` se encuentra en la dirección `0x00020000`. La pila se alojará en la página `0x0002A000` y será mapeada con *identity mapping*. Esta tarea ocupa 2 paginas de 4KB y está compilada para ser ejecutada desde la dirección `0x40000000`. Además la misma debe compartir el mismo `CR3` que el *kernel*.

c) Completar el resto de las entradas del arreglo de las `TSS` definidas con los valores correspondientes a las tareas que correrá el sistema. A cada tarea le corresponden dos `TSS`, una para el código de la tarea y otra para la ejecución de la función "bandera". Para ambas `TSS` el contexto de ejecución será el mismo, dependerá de la tarea que nada explote en pedazos. El código de las tareas se encuentra a partir de la dirección `0x00010000` ocupando dos páginas de 4kb cada una. El mismo debe ser mapeado a partir de la dirección `0x40000000`. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea según indica la figura 5, al igual que para la dirección de la pila en la función "bandera". Para el mapa de memoria se debe construir uno nuevo para cada tarea utilizando la función `mmu_inicializar_dir_usuario`; tanto la función "bandera" como el código de la tarea deben compartir el mismo mapa de memoria. Tener en cuenta que ambas van a utilizar distinto espacio de pila de nivel 0. Para este propósito se recomienda utilizar una sola pagina en nivel de kernel.

d) Completar la entrada de la `GDT` correspondiente a la `tarea_inicial`.

e) Completar la entrada de la `GDT` correspondiente a la tarea `Idle`.

f) Completar el resto de las entradas de la `GDT` para cada una de las entradas del arreglo de `TSS`s de las 8 tareas que se ejecutarán en el sistema.

g) Escribir el código necesario para ejecutar la tarea `Idle`, es decir, saltar intercambiando las `TSS`, entre la `tarea_inicial` y la tarea `Idle`.

Nota: En `tss.c` está definido un arreglo llamado `tss` que contiene las estructuras `TSS`. Trabajar en `tss.c` y `kernel.asm` .

4.7. Ejercicio 7

a) Construir una función para inicializar las estructuras de datos del *scheduler* (arreglo de tareas y arreglo de funciones "bandera").

b) Crear la función `sched_proximo_indice()` que devuelve el índice en la `GDT` de la próxima tarea a ser ejecutada. Ésta debe tener en cuenta lo descripto en la sección 3.2.

c) Crear la función `sched_proxima_bandera()` que devuelve el índice en la `GDT` de la próxima bandera a ser ejecutada. Ésta debe tener en cuenta lo descripto en la sección 3.2.

d) Modificar la rutina de la interrupción `0x50`, para que implemente los tres servicios del sistema según se indica en la sección 3.1.1.

e) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. El intercambio se realizará según indique la función `sched_proximo_indice()`.

f) Cuando la cantidad de intercambios entre tareas llegue a 3, se deberá llamar a todas las funciones bandera. Escribir el código necesario para que se respete este comportamiento.

g) Modificar las rutinas de excepciones del procesador para que impriman el problema que se produjo en pantalla, desalojen a la tarea que estaba corriendo y corran la próxima, indicando en pantalla porque razón fue desalojada la tarea en cuestión. Tener en cuenta que las "banderas" también pueden generar problemas.

Nota: Se recomienda construir funciones en C que ayuden a resolver problemas como convertir direcciones a posiciones de la arena.

4.8. Ejercicio 8 (optativo)

a) Crear un navío (tarea) propio que navegue a muerte contra los navíos creados por los docentes. Para esto pueden editar el código de la tarea 1 a gusto.

   La tarea debe tener las siguientes características

   - No ocupar más de 8 kb (tener en cuenta la pila, figura 5).
   - Tener como punto de entrada la dirección cero.
   - Contener en la última posición de memoria el puntero a la función "bandera".
   - Estar compilada para correr desde la dirección `0x400000000000`.
   - Utilizar solo los servicios presentados en el trabajo práctico.

b) Explicar en pocas palabras qué estrategia utilizaron en su navío mortal en términos de "defensa" y "ataque".

c) Si consideran que su navío mortal es capaz de enfrentarse contra los navíos del resto de sus compañeros, pueden enviar el **binario** a la lista de docentes indicando los siguientes datos,

   - Nombre del navío, *ej: "El Perla Morada"*
   - Características letales, *ej: Se mueve rápidamente*
   - Sistema de defensa, *ej: Es prácticamente un rompehielos*

   Se realizará una competencia a fin de cuatrimestre con premios en/de chocolate para los primeros puestos.

d) (Optativo) Mencionar en que intersección de calles y barrio de la Capital Federal de Buenos Aires, se encuentra el ancla de la figura 2.

5. Entrega

La resolución de los ejercicios se debe realizar gradualmente. Dentro del archivo `kernel.asm` se encuentran comentarios (que muestran las funcionalidades que deben implementarse) para resolver cada ejercicio. También deberán completarse el resto de los archivos según corresponda.

A diferencia de los trabajos prácticos anteriores, en este trabajo está permitido modificar cualquiera de los archivos proporcionados por la cátedra, o incluso tomar libertades a la hora de implementar la solución; siempre que se resuelva el ejercicio y cumpla con el enunciado. Parte de código con el que trabajen está programado en ASM y parte en C, decidir qué se utilizará para desarrollar la solución, es parte del trabajo.

Se deberá entregar un informe que describa **detalladamente** la implementación de cada uno de los fragmentos de código que fueron construidos para completar el kernel. En el caso que se requiera código adicional también debe estar descripto en el informe. Cualquier cambio en el código que proporcionamos también deberá estar documentado. Se deberán utilizar tanto pseudocódigos como esquemas gráficos, o cualquier otro recurso pertinente que permita explicar la resolución. Además se deberá entregar en soporte digital el código e informe; incluyendo todos los archivos que hagan falta para compilar y correr el trabajo en Bochs.

La fecha de entrega de este trabajo es **10-11** y deberá ser entregado a través de la página web en un solo archivo comprimido en formato `tar.gz`, con un límite en tamaño de 10Mb. El sistema sólo aceptará entregas de trabajos hasta las **16:59** del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.