Trabajo Práctico 3
System Programming - `Jaurías`

Organización del Computador 2

Segundo Cuatrimestre de 2015

1. Objetivo

Este trabajo práctico está compuesto por 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 hasta 16 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. Algunas tareas podrán ser cargadas en el sistema dinámicamente por medio del uso del teclado.

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*)
- `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 para imprimir por pantalla (preferentemente NO usar, usar las funciones de screen).
- `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`.
- `idle.asm` - código de la tarea `Idle`.
- `game.h`, `game.c`, `jugador.c`, `perro.c` - implementación de la lógica del juego.
- `game_isr.c` - handlers de alto nivel de las interrupciones/syscalls (teclado, reloj, mover, cavar, etc).
- `tareaA1.c`, `tareaA2.c`, `tareaB1.c`, `tareaB2.c` - código de las tareas (*dummy*).
- `syscalls.h` - interfaz a utilizar en C para hacer llamados al sistema.

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 propia implementación.

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. Reglas del juego

En este trabajo práctico vamos a implementar un juego de dos jugadores. Cada jugador tendrá la posibilidad de soltar perros hacia el territorio. Estos perros serán tareas del sistema que se moverán por la memoria en busca de huesos para consumir. Como es habitual, los huesos estarán enterrados, por lo tanto los perros tendrán que desenterrarlos. Es decir los perros serán tareas que se moverán por la memoria buscando huesos para desenterrar. Una vez desenterrados deberán llevarlos a la cucha. Denominamos "escondite" a una posición del mapa donde hay huesos enterrados. Además, los jugadores (que no son tareas), se irán moviendo en el terreno y darán órdenes a su respectiva jauría.

Cada jugador podrá tener un máximo de 8 perros en juego de distintas *razas*. Para obtener huesos el perro, llamémosle *Bobby*, deberá moverse hasta un escondite y deberá desenterrarlos. Una vez desenterrados, Bobby deberá llevarlos a la cucha. Para orientarse, Bobby deberá olfatear, y así saber en que dirección está el hueso más cercano. Además, de vez en cuando Bobby podrá escuchar órdenes de su amo. Los jugadores, que estarán parados en alguna posición del terreno, podrán emitir órdenes mediante la pulsación de algunas teclas. Quedará librado a la voluntad de cada perro aceptar o ignorar dichas órdenes.

Para cavar, Bobby deberá pararse justo sobre un escondite y luego cavar. Cada vez que cave obtendrá un hueso, hasta un maximo de 10 o hasta que los huesos de ese escondite se hayan agotado. Bobby deberá entonces llevar los huesos a la cucha, donde se descargarán solos con el correr del tiempo. Una vez que Bobby descarga todos sus huesos se duerme, liberando su lugar para otro perro.

El objetivo del juego será obtener la mayor cantidad de huesos posibles. El juego termina cuando se agotan los huesos. Gana el jugador cuyos perros llevaron más huesos a la cucha. Es posible llegar al caso en que todos los slots estén ocupados y no se puedan lanzar más perros, y que estos no sean capaces de desenterrar los huesos. En este caso se da por terminado el juego luego de un tiempo prudencial en que no pasa nada (no hay cambios de puntajes).

3.1. Las razas y las órdenes del amo

Cada raza de perro se corresponderá con un programa que modele su comportamiento. Se pueden implementar 2 razas distintas por jugador[1]. Cada raza modelará cómo interpretar las órdenes del amo. Por ejemplo, si *Bobby* fuera de una raza holgazana podría no darle importancia a las órdenes, pero si fuera de una raza de ataque podría interpretar una orden como atacar a un perro del otro jugador.

El amo, es decir el jugador podrá darle tres tipos de órdenes a su jauría: A, B y C. Dependerá de cada raza de perro interpretar la orden. El perro sólo podrá conocer cual fue la última orden y en que posicion estaba el jugador cuando la dio.

4. Reglas de implementación

El sistema correrá un máximo de 16 tareas concurrentemente, cada una de las cuales representará a un perro. El código de los perros puede verse en los archivos `tarea*.c`. Estas tareas podrán realizar varias acciones, algunas de las cuales requerirán control del sistema, por lo que serán implementadas como *syscalls*: moverse, cavar, olfatear y recibir ordenes. Además, existirá una tarea *idle*, también de nivel usuario, para los momentos en que no haya que correr el código de ningún perro.

Cada jugador lleva asignada una posición del *mapa* donde está la cucha de sus perros. De ahí saldrán, y allí deberán volver. Cada posición del mapa corresponde a una página de 4kb de memoria. Este mapa corresponderá a un área de memoria física de 80×44 paginas de 4kb.

Los jugadores se dibujarán en el mapa (tienen coordenadas x e y), pero no ocupan memoria ni son tareas. Su única función será moverse por el mapa lanzando perros y emitiendo órdenes.

[1] La cátedra proporciona 1 para testeo. La idea es que puedan crear las suyas propias para la guerra de tareas

4.1. Organización de la memoria física

Al ejecutar make, se ejecutará una cadena de compilación, linkeo y empaquetado. Primero, todo el código escrito por nosotros se compilará y así se obtendrá un montón de archivos objeto (.o). A partir de ellos, se creará un gran archivo binario del kernel (`kernel.bin`), que luego se copiará a una imagen de diskette en formato *FAT12*, que será el *boot disk*. Luego de la inicialización del sistema, la memoria se encontrará como se ve en la figura 1.

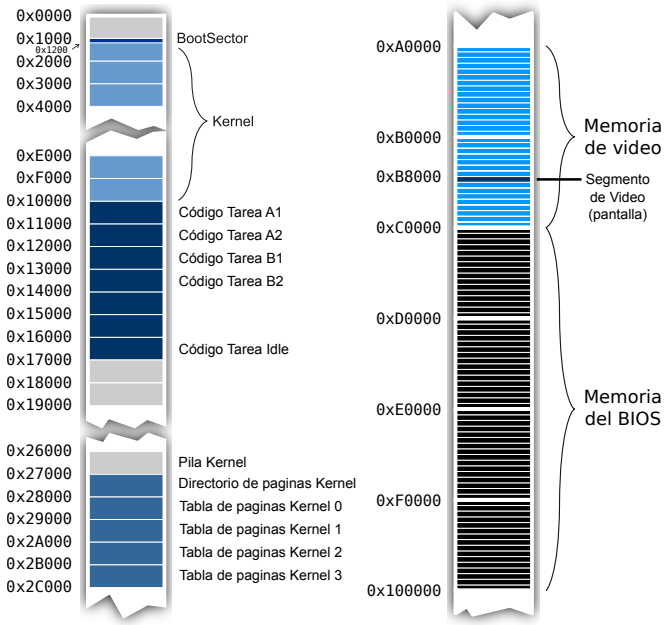

Figura 1: Mapa de la organización de la memoria física del *kernel*

4.2. Organización de la memoria virtual

Para poder correr en un entorno protegido, la memoria de las tareas será isolada mediante paginación. Esto implica que lo que cada una vea en la memoria no será exactamente igual a como se almacena físicamente. Cada tarea lleva mapeadas todas las páginas correspondientes a las posiciones del mapa que a visitado el perro, en sólo-lectura. Inicialmente sólo lleva mapeada la posición correspondiente a la cucha. Estas páginas se mapearan a partir de la dirección 0x800000 correspondiendo en orden con las direcciones físicas del mapa, que empieza en la posición física 0x500000. Además, para cada tarea, la misma página física correspondiente a su posición actual en el mapa irá mapeada como código y datos con permisos de lectura/escritura en la dirección 0x401000. Por último, para colaborar entre sí, todos los perros de un mismo jugador llevan mapeada una misma página compartida en la dirección 0x400000.

Si un perro quiere moverse, debe pedírselo al sistema operativo, el cual copiará su código y pila a la página donde se mueva. Además, las tareas perros tendrán en todo momento en su dirección virtual 0x400000 una página extra de lectura-escritura, que apuntará a la memoria física de su ubicación en el mapa. Esto le brindará espacio para su código y pila.

Una vez concluída la inicialización del sistema, el kernel -es decir, el código del sistema

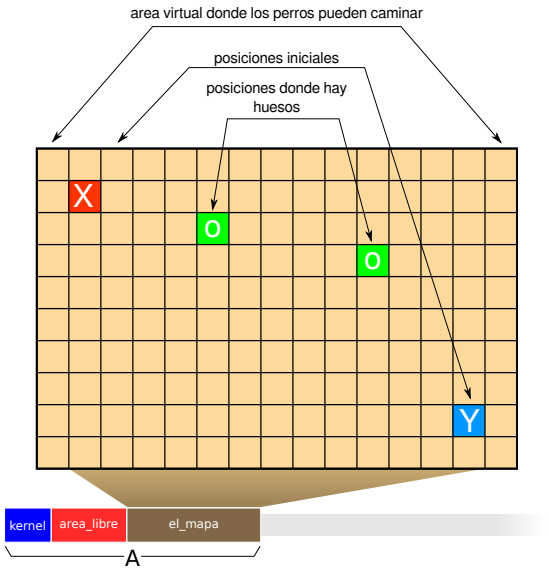

Figura 2: Mapa inicialmente

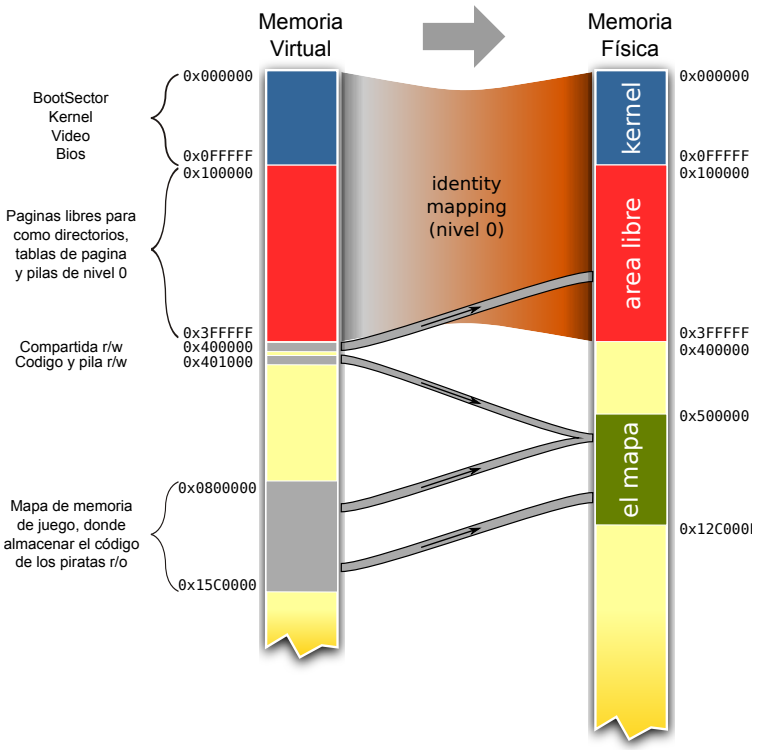

Figura 3: Mapa de memoria de la tarea

operativo-, se convierte en algo *pasivo*. Es decir, nunca va a correr, a menos que alguien lo invoque. Esto *siempre* ocurrirá en el contexto de una interrupción: cuando se presiona una tecla, con un tick de reloj, o cuando una tarea falla o hace un syscall. Durante el manejo de una interrupción el procesador pasa a un nivel de privilegio más alto, pero no cambia de tarea, por lo que el kernel verá a la memoria de la misma forma que las tareas, aunque con mayores privilegios. Entonces, para que el procesador pueda encontrar al código del kernel, tanto la tarea idle como cada una de las tareas perros tiene mapeadas las áreas de *código kernel* y *libre* con *identity mapping* en nivel 0. La memoria física del *mapa* no está mapaeada por defecto, por lo que cuando el kernel quiera escribir allí deberá mapear antes la página correspondiente. No obstante, el *kernel* puede escribir en cualquier posición del área *libre* desde cualquier perro sin tener que mapear esta área.

En la figura 3 se puede ver la posición de las páginas y donde deberán estar mapeadas como direcciones virtuales dentro del área de memoria del perro.

La memoria libre será administrada de forma muy simple. Se tendrá un contador de páginas libres a partir de las que se solicita una nueva. Siempre se aumenta en cantidad de páginas usadas y nunca se liberan las páginas pedidas.

4.3. Acciones del jugador

Las acciones que puede llevar a cabo el jugador se activarán mediante el teclado. Esta funcionalidad será resuelta en la interrupción de teclado llamando a funciones implementadas para la lógica del juego. Esta parte del sistema correrá en nivel supervisor y por lo tanto no podrá ser afectada por las tareas perros. A continuación se listan las acciones y las teclas usadas por cada jugador:

Lanzar un nuevo perro 

Deberá realizar varias acciones: inicializar el mapeo de memoria del perro, copiar el código desde el kernel al mapa (en la posición de la cucha), inicializar el stack y una tss para finalmente agregar la tarea al scheduler. Si ya hay un perro en la cucha, la acción de lanzar un nuevo perro no tendrá ningún efecto. La tarea recibe como argumentos la dirección de salida (x,y).

Moverse por el mapa 

Simplemente debe acualizar la posición del jugador en la pantalla.

Dar una orden 

Debe guardar el tipo y posicion de la última orden, por si algún perro del jugador pregunta.

4.4. Acciones de perros ¡guau! (Servicios del Sistema)

El sistema provee un único servicio que corresponde a todas las acciones que pueden hacer los perros: moverse, cavar, olfatear y averiguar órdenes. Este servicio será implementado como una *system call* mapeada a la interrupción `0x46`. Su primer parámetro será el tipo de pedido requerido, por medio del registro `EAX`. El segundo parámetro, en caso de ser necesario, se pasará por `ECX`. A continuación se presenta la tabla para la codificación de los pedidos:

`0x1 - Moverse.` Recibe en ECX el código de dirección, que puede ser: 4, arriba — 7, abajo — 10, derecha — 13, izquierda. Deberá copiar el código de la tarea a la nueva ubicación, siempre y cuando sea posible: debe chequearse que no se salga del mapa y que no haya una tarea del mismo jugador en el lugar destino. Deberá mapear la nueva posición del mapa a esa tarea.

`0x2 - Cavar.` En caso de estar sobre un escondite aumenta en 1 la cantidad de huesos del perro y disminuye en 1 la cantidad de huesos del escondite. La cantidad de huesos en cada escondite están dados por una variable global llamada `huesos` en `game.h`. El perro puede llevar hasta 10 huesos, luego de eso el syscall debe ignorar el pedido.

`0x3 - Olfatear.` El tercer caso del syscall le permite a una tarea perro conocer la dirección del hueso más cercano (es decir, izquierda, derecha, arriba o abajo).

`0x4 - Recibir orden.` Esta función del syscall le permite a una tarea perro conocer la última orden emitida por el jugador. Cada raza de perro interpretará libremente la orden. El sistema debe devolver un par (x,y) y un numero indicando el numero de orden (1, 2 o 3). Para poder devolver 3 valores en uno, deberá codificarlos de la siguiente manera: `id\_orden << 16 | y << 8 | x`.

Por último, es fundamental tener en cuenta que una vez llamado el servicio, el *scheduler* se encargará de desalojar a la tarea que lo llamó para dar paso a la próxima tarea. Este mecanismo será detallado mas adelante.

4.5. Scheduler

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

Como el sistema tiene dos jugadores, los perros de cada uno de los dos jugadores serán anotados en dos conjuntos distintos. El *scheduler* se encargará de repartir el tiempo entre los dos conjuntos sin importar la cantidad de perros que tenga cada uno. Esto quiere decir que por cada *tick* de reloj se ejecutará un perro de cada jugador por vez.

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 porque se llamó de forma incorrecta a un servicio.

Cualquier acción que realice una tarea de forma incorrecta será penada con el desalojo de dicha tarea del sistema, es decir la "muerte" del perro.

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 al 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.

Por cada tick de reloj deberá avisarse al juego, para que modifique los relojes en pantalla y además chequee si el perro actual está en la cucha (en cuyo caso se suma 1 punto al jugador y si el perro ya descargó todo se elimina a esa tarea).

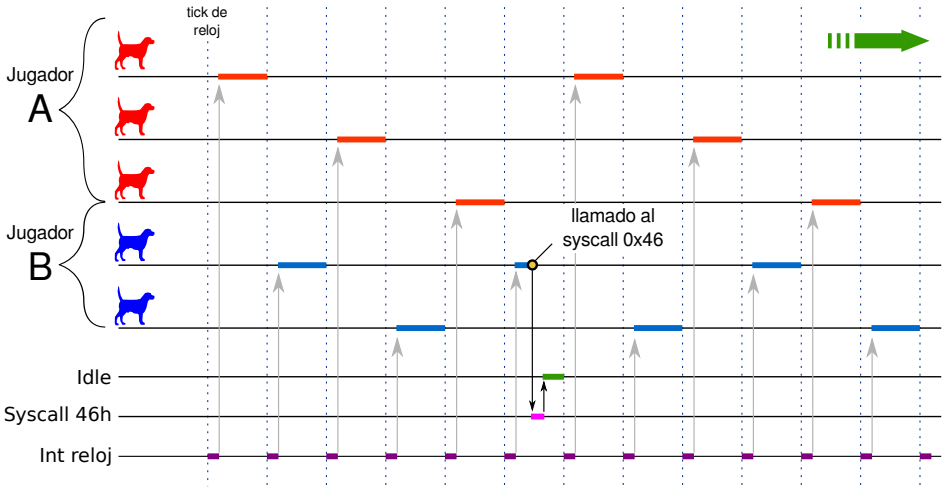

Figura 4: Ejemplo de funcionamiento del *Scheduler*

Inicialmente, la primera tarea en correr es la `Idle`. Luego, en algún momento alguno de los jugadores lanzará algún perro, lo que implicará que en el próximo *tick* de reloj se comience la ejecución de esta tarea. La misma correrá hasta que termine su tiempo en el próximo *tick* de reloj o la tarea intente llamar al servicio del sistema (moverse); de ser así, será desalojada y el tiempo restante será asignado a la tarea `Idle`. En la figura 4 esto sucede con el primer perro del jugador B.

4.6. Estructuras para la administración del sistema

El sistema tendrá que almacenar estructuras de datos necesarias para salvar información de las tareas y del juego. Para esto se utilizará una copia del contexto de cada tarea correspondiente a una *tss*.

Además el sistema tendrá control del juego, para esto se deberán salvar algunos datos:

- Información del jugador, como su posición, su puntaje, etc
- Posición de cada perro dentro del mapa
- Perro/tarea que está siendo actualmente ejecutada y una forma de acceder a la siguiente tarea por ser ejecutada
- Tareas en ejecución y slots libres donde correr nuevas tareas
- Páginas mapeadas por cada tarea

Es decisión de implementación cómo almacenar la información antes listada. Sin embargo, la memoria en este sistema será administrada de forma muy simple como fue explicado anteriormente.

4.7. Pantalla

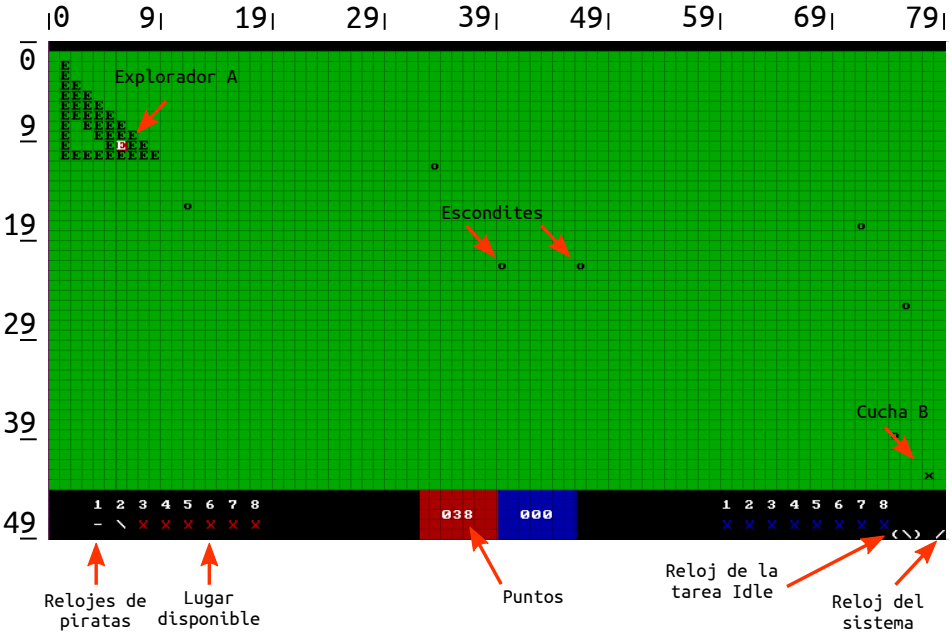

Figura 5: Pantalla de ejemplo[2]

La pantalla presentará un mapa donde se producirá la acción e información del estado de cada jugador y perros en el sistema.

La figura 5 muestra una imagen ejemplo de pantalla indicando qué datos deben presentarse de forma mínima. Se recomienda implementar funciones auxiliares que permitan imprimir datos en pantalla de forma cómoda. No es necesario respetar la forma de presentar los datos en pantalla, se puede modificar la forma, no así los datos en cuestión.

4.8. Modo debug

El sistema deberá responder a una tecla especial en el teclado, la cual activará y desactivará el modo debugging. La tecla para tal proposito es la "y". En este modo se deberá mostrar en pantalla la primera excepción capturada por el procesador junto con un detalle de todo el estado del procesador como muestra la figura 6. Una vez impresa en pantalla esta excepción, el juego se detendrá hasta presionar nuevamente la tecla "y" que mantendrá el modo de debug pero borrará la información presentada en pantalla por la excepción. La forma de detener el juego será instantaneamente, al retomar el juego se esperará hasta el próximo ciclo de reloj en el que se decida cuál es la proxima tarea a ser ejecutada. Se recomienda hacer una copia de la pantalla antes de mostrar el cartel con la información de la tarea.

[2]En este ejemplo faltan dibujar los 2 jugadores, que deberían estar en algun lugar del mapa

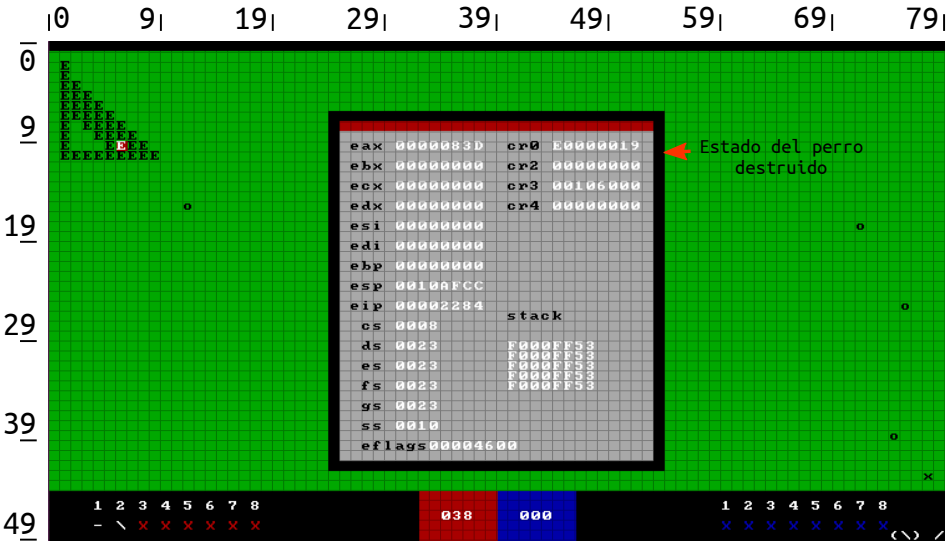

Figura 6: Pantalla de ejemplo de error

5. Ejercicios

Se detallan aquí los pasos necesarios para completar la implementación del trabajo. Algunas secciones del mismo requerirán escribir código ensamblador. En otras partes no hará falta y podrá usarse C. Recomendamos *fuertemente* evitar escribir código ensamblador para todo, y usar C lo más posible. Antes de comenzar a implementar cada sección de código se recomienda preguntarse, **¿es realmente necesario escribir este código en ensamblador o puedo hacerlo en C?** en caso de que la respuesta sea que se requiere assembly, intentar crear una interfaz C-Assembler donde el código C llame a funciones que luego se implementen en ASM y viceversa.

Mucha de la funcionalidad relativa al juego, aquella que no está relacionada con los conceptos de la materia, se entrega ya implementada para facilitar el trabajo. Algunas secciones de código se encuentran parcialmente implementadas, y otras deben escribirse desde cero. Queda a criterio de cada grupo respetar el diseño sugerido o modificarlo totalmente, tanto a nivel código como a nivel estético (siempre y cuando se cumpla con todas las funcionalidades pedidas).

5.1. Ejercicio 1 - Pasaje a modo protegido y segmentación

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 500MB de memoria. En la `GDT`, por restricción del trabajo práctico, las primeras 7 posiciónes se consideran utilizadas y no deben utilizarse. El primer índice que deben usar para declarar los segmentos, es el 8 (contando desde cero).

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

c) Para probar el funcionamiento de la segmentación, agregar a la `GDT` un segmento adicional

que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*. Escribir una rutina que pinte el extremo superior izquierdo de la pantalla utilizando este segmento. A partir del próximo ítem y para los próximos ejercicios, acceder la memoria de video directamente por medio del segmento de datos de 500MB.

d) Escribir una rutina que se encargue de limpiar la pantalla[3] y pintar en el área de *el_mapa* un fondo de color (sugerido gris), junto con las dos barras inferiores para cada uno de los jugadores (sugerido rojo y azul). Se recomienda para este ítem implementar las funciones señaladas como auxiliares en `screen.h`.

Comandos útiles: `info gdt`. 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`.

5.2. Ejercicio 2 - Interrupciones básicas

a) Completar las entradas necesarias en la `IDT` para asociar diferentes rutinas a todas las excepciones del procesador. Para tal fin, aprovechar la macro `ISR` en `isr.asm`, y la macro `IDT_ENTRY` en `idt.c`. 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`.

Comandos útiles: `info idt`

5.3. Ejercicio 3 - Paginación básica

a) Escribir una rutina que se encargue de limpiar el *buffer* de video y pintarlo como indica la figura 5. 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 una rutina encargada de desmapear páginas de memoria
`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`.

c) Utilizando la función anterior, escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el *kernel* (`mmu_inicializar_dir_kernel`). Es decir, se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x003FFFFF`, como ilustra la figura 3. El directorio de páginas a inicializar se encuentra en la dirección `0x27000`, y las tablas de páginas según muestra la figura 1.

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

d) Completar el código necesario para activar paginación. Verificar que el sistema sigue funcionando luego de que la paginación sea activada llamando a una rutina que imprima el nombre del grupo en pantalla. Debe estar ubicado en la primer línea de la pantalla alineado a derecha.

e) Escribir una rutina encargadas de mapear páginas de memoria
`mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

f) Probar la función anterior desmapeando la última página del kernel (0x3FF000).

Comandos útiles: `info tab`

5.4. Ejercicio 4 - Paginación dinámica

a) Escribir una rutina (`inicializar_mmu`) que se encargue de inicializar las estructuras globales necesarias para administrar la memoria en el area libre (un contador de paginas libres).

b) Escribir una rutina (`mmu_inicializar_memoria_perro`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3. La rutina debe copiar el código de la tarea a su área asignada, es decir la posición indicada por el jugador dentro de *el_mapa* y configurar su directorio segun se indica en 4.2. Nota: puede ser conveniente agregar a esta función otros parámentros que considere necesarios.

c) Inicializar el mapa de memoria de una tarea perro e intercambiarlo con el del *kernel*, luego cambiar el color del fondo del primer caracter de la pantalla y volver a la normalidad. Este ítem no debe estar implementado en la solución final.

Nota: 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.

5.5. 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 una a la interrupción de software `0x46`.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `game_atender_tick`. 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 `screen_actualizar_reloj_global` está definida en `screen.h`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de las teclas a utilizar en el juego, se presente la misma en la esquina superior derecha de la pantalla.

d) Escribir la rutina asociada a la interrupción `0x46` para que modifique el valor de `eax` por $0x42$. Posteriormente este comportamiento va a ser modificado para atender el servicio del sistema.

SUGERENCIA: Construir la rutina de assembler lo mas corta posible: guardar los registros, llamar a `game_tick`, restaurar los registros y salir de la interrupción.

5.6. Ejercicio 6

a) Definir las entradas en la `GDT` que considere necesarias para ser usadas como descriptores de `TSS`. Minimamente, una para ser utilizada por la `tarea_inicial` y otra para la tarea Idle.

b) Completar la entrada de la `TSS` de la tarea Idle con la información de la tarea `Idle`. Esta información se encuentra en el archivo `TSS.C`. La tarea `Idle` se encuentra en la dirección `0x00016000`. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con *identity mapping*. Esta tarea ocupa 1 pagina de 4KB y debe ser "mapeada" con *identity mapping*. Además la misma debe compartir el mismo `CR3` que el *kernel*.

c) Construir una función que complete una `TSS` libre con los datos correspondientes a una tarea. El código de las tareas se encuentra a partir de la dirección `0x00010000` ocupando una pagina de 4kb cada una según indica la figura 1. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea, la misma crecerá desde la base de la tarea. Recordar que las tareas asumen que se habrán apilado sus 2 argumentos y luego una dirección de retorno. Para el mapa de memoria se debe construir uno nuevo utilizando la función `mmu_inicializar_dir_perro`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva pagina libre a tal fin.

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

g) Modificar la rutina de la interrupción `0x46`, para que implemente los servicios según se indica en la sección 4.4, sin desalojar a la tarea que realiza el syscall.

h) Ejecutar una tarea `perro` manualmente. Es decir, crearla y saltar a la entrada en la gdt de su respectiva `TSS`

Nota: En `tss.c` están definidas las `tss` como estructuras `TSS`. Trabajar en `tss.c` y `kernel.asm` .

5.7. Ejercicio 7

a) Construir una función para inicializar las estructuras de datos del *scheduler*.

b) Crear la función `sched_proxima_a_ejecutar()` que devuelve el índice de la próxima tarea a ser ejecutada. Construir la rutina de forma que devuelva una tarea de cada jugador por vez según se explica en la sección 4.5.

c) Crear una función `sched_atender_tick()` que llame a `game_atender_tick()` pasando el numero de tarea actual y luego devuelva el indice en la gdt al cual se deberá saltar. Reemplazar el llamado a `game_atender_tick` por uno a `sched_atender_tick` en el handler de la interrupción de reloj.

d) Modificar la rutina de la interrupción `0x46`, para que implemente los servicios según se indica en la sección 4.4.

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_proxima_a_ejecutar()`.

f) Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y corran la próxima.

g) Implementar el mecanismo de debugging explicado en la sección 4.8 que indicará en pantalla la razón del desalojo de una tarea.

5.8. Ejercicio 8 (optativo)

a) Crear un conjunto de 2 tareas perros. Los mismos deberán respetar las restricciones del trabajo práctico, ya que de no hacerlo no podrán ser ejecutados en el sistema implementado por la cátedra.

   Deben cumplir:

   - No ocupar más de 4 kb cada uno (tener en cuenta la pila).
   - Tener como punto de entrada la dirección cero.
   - Estar compilado para correr desde la dirección `0x0401000`.
   - Utilizar el único servicio del sistema (`0x46`).

   Explicar en pocas palabras qué estrategia utiliza cada uno de los perros, o en su conjunto en términos de "defensa" y "ataque".

b) Si consideran que sus tareas pueden hacer algo mas que completar el primer ítem de este ejercicio, y tienen a un audaz campión que se atreva a enfrentarse en el campo de batalla perro, entonces pueden enviar el **binario** de sus tareas a la lista de docentes indicando los siguientes datos,

   - Nombre del campion (Alumno de la materia que se presente como "jugador")
   - Nombre de cada uno de las tareas perro
   - Estrategia de exploracion y supervivencia (es decir, como se sobrevivirán en el cruel mundo perro)

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

c) Pelicula y Video Juego favorito sobre Perros.

6. Entrega

Este trabajo práctico esta diseñado para ser resuelto de forma gradual.

Dentro del archivo `kernel.asm` se encuentran comentarios (que muestran las funcionalidades que deben implementarse) para resolver cada ejercicio. También deberán completar el resto de los archivos según corresponda. En ellos encontrarán un conjunto de funciones declaradas pero no definidas o incompletas. Las mismas se ofrecen como guía y ayuda para el desarrollo del trabajo.

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 de Noviembre** 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 6.28318530718Mb. 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.