Trabajo Práctico 3
System Programming - `Tierra pirata`

Organización del Computador 2

Primer Cuatrimestre de 2015

1. Objetivo

Este trabajo práctico 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 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 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 y lógica del juego.
- `syscalls.h` - interfaz utilizar en C los llamados al sistema.
- `tareaAE.c`, `tareaAM.c`, `tareaBE.c`, `tareaBM.c` - código de las tareas (*dummy*).
- `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 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.

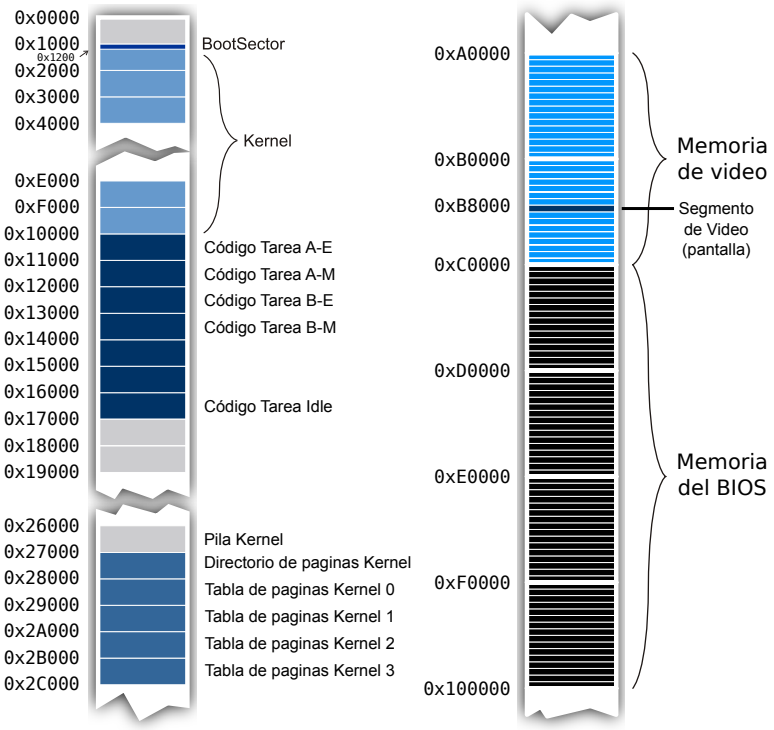

Figura 1: Mapa de la organización de la memoria física del *kernel*

3. Tierra pirata

En este trabajo práctico vamos a implementar un juego de dos jugadores (`A` y `B`). Cada jugador tendrá la posibilidad de lanzar piratas exploradores hacia el territorio. Estos exploradores serán tareas del sistema que se moverán por la memoria en busqueda de botines para consumir. Como siempre con los piratas, los botines están enterrados, por lo que necesitaremos piratas mineros que los desentierren. Estos piratas también serán tareas que se moverán por la memoria.

Cada jugador podrá tener un máximo de 8 piratas en juego entre exploradores y mineros. Cuando un pirata explorador detecte un botín, se llamará a un pirata minero que deberá ir a cavar para desenterrarlo. Cuando un minero termina su trabajo, libera su slot, a diferencia de los exploradores, que nunca terminan. Los piratas son capaces de algunas cosas, entre ellas, moverse. Cuando un pirata decide moverse, llama al sistema que mueve toda su memoria a una nueva posición. Además, un pirata puede preguntar cual es su posición en el mapa, la de sus compañeros y también cavar, si es minero. En la figura 2 se da cuenta del llamado al sistema que les permite realizar estas acciones.

Para poder cavar, un pirata deberá pararse justo sobre el botín y luego cavar. Por cada vez que cave obtendrá una moneda de oro, hasta que el botín se haya agotado. El objetivo será que el jugador obtenga la mayor cantidad de monedas posibles. El juego termina cuando se agotan los botines. Gana el jugador que obtuvo más monedas de oro. Es posible llegar al caso en que todos los slots estén ocupados y no se puendan lanzar mas piratas, y que estos no sean capaces de desenterrar las monedas. 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. Piratas (Tareas)

El sistema correrá un máximo de 16 tareas concurrentemente. Las mismas podrán realizar varias acciones, que serán implementadas como un único servicio del sistema.

Cada jugador tendrá asignada inicialmente un area pequeña (3x3) del *mapa*. 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.

Las tareas piratas partirán siempre del puerto del jugador, uno en cada extremo del mapa. Luego se moverán hacia donde lo deseen siguiendo su propia voluntad. A su memoria virtual se irán mapeando secciones del mapa, a medida que los piratas exploradores las vayan descubriendo. Al moverse un explorador, se mapearán las páginas a su alrededor a *todas* las tareas del jugador. Las páginas se mapearán como sólo-lectura. Si un pirata -cualquiera sea- decide moverse, el sistema operativo copiará su código a la página donde se mueva. Además, las tareas piratas 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.

3.1.1. Acciones piratas ¡arrrr! (Servicios del Sistema)

El sistema provee un único servicio que corresponde a todas las acciones que pueden hacer los piratas: moverse, preguntar su posición y cavar. Este servicio será implementado como una *system call* mapeada a la interrupción `0x46`. Su primer parámetro pasado será el tipo pedido realizado, 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 desplazamientos:

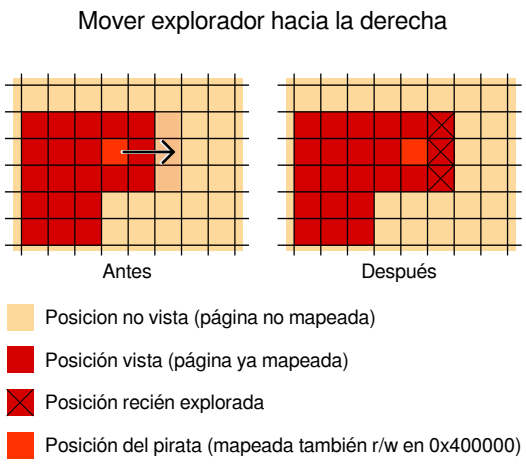

Figura 2: El Juego

`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 en caso de ser un minero la posición ya debe haber sido mapeada. En caso de ser un explorador deberá además mapear las posiciones de alrededor (según la figura 2), tanto para esa tarea como para las otras del jugador. En caso de descubrir botines deberá crear una tarea minero por cada uno de ellos, pasando la dirección `(x,y)` donde se encuentra a la tarea minero. Se considera que un botin fue descubierto cuando la posición en que se encuentra es descubierta por un explorador del jugador. Si no hay slots disponibles deberá guardarlos para cuando alguno se libere.

`0x2 - Cavar.` En caso de estar sobre un botín aumenta en 1 el puntaje del jugador y disminuye en 1 la cantidad de monedas de esa posición. Los botines vienen dados por una variable global llamada `botines` en `game.h`. Si no hay más monedas, el sistema terminará a la tarea pirata minero y se liberará su slot.

`0x3 - Posicion.` El tercer caso del syscall deberá retornar un valor. Este valor corresponderá a la posición del pirata pasado como parámetro. El parámetro puede ser de 0 a 7, para indicar numéricamente un pirata del mismo jugador, o -1 para obtener la del propio pirata. La posición se devolverá a la tarea llamadora en el registro `EAX` codificada de la siguiente manera: `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.

3.1.2. Nuevo Pirata y lógica de juego

Una funcionalidad que se debe destacar es la posibilidad que brinda el sistema de lanzar nuevos piratas. Esta parte del sistema será implementada en nivel supervisor y por lo tanto no podrá ser afectada por las tareas piratas. La lógica del juego corresponde a solucionar los siguientes problemas:

- Lanzar un nuevo pirata:
  Esta funcionalidad será resuelta a nivel del scheduler que generará una nueva tarea. El llamado será desde la interrupción de teclado.

- Controlar teclas:
  Esta funcionalidad será resuelta en la interrupción de teclado llamando a funciones implementadas para la lógica del juego.

Las teclas que utilizarán cada uno de los jugadores se presentan en la siguiente tabla:

Acción	Jugador A (izquierda)	Jugador B (derecha)	Descripción
⇑	LShift	RShift	Lanzar pirata explorador

3.1.3. Organización de la memoria y procesos

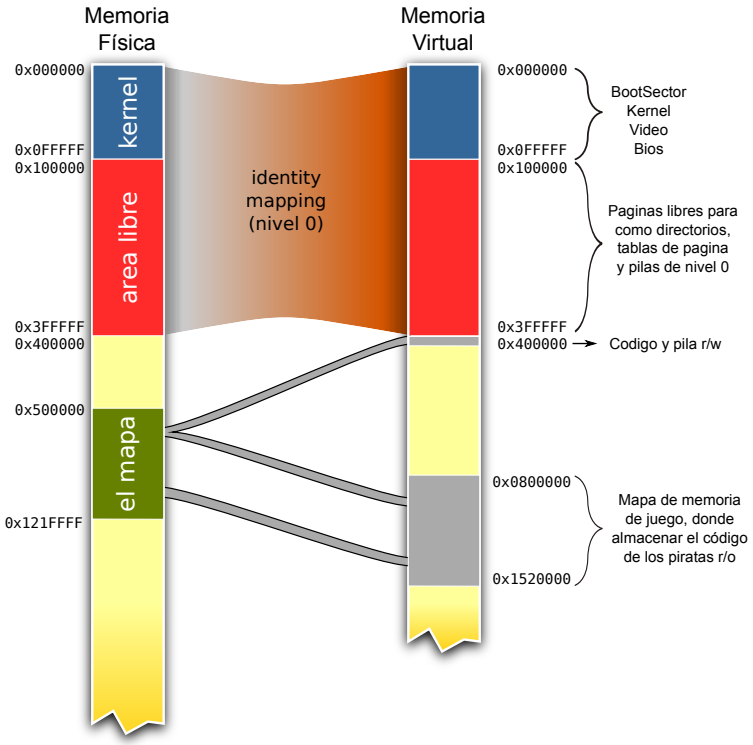

Figura 3: Mapa de memoria de la tarea

Luego de la inicialización del sistema, lo que comúnmente denominamos kernel -el código del sistema operativo-, corre *siempre* en el contexto de una interrupción. Esto es, cuando se presiona una tecla, con un tick de reloj, o cuando una tarea falla o hace un syscall. Es decir que el kernel ve a la memoria de la misma forma que las tareas (ya sea idle o piratas), aunque con mayores privilegios. Tanto la tarea idle como cada una de las tareas piratas tiene

mapeadas las áreas de *código kernel* y *libre* con *identity mapping* en nivel 0. Esto no es así para el área del *mapa*. Esto obliga al *kernel* a mapear el *mapa* cada vez que quiera escribir en él. No obstante, el *kernel* puede escribir en cualquier posición de el área *libre* desde cualquier pirata sin tener que mapear esta área.

Ademaś, todas las tareas de un mismo jugador llevan mapeadas todas las páginas del mapa descubierto por los exploradores de ese jugador en sólo-lectura. 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. Por último, cada tarea pirata lleva mapeada una página para datos y código en nivel 3, que corresponde al código de la tarea, con permisos de lectura/escritura. 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 pirata.

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.

3.2. 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* minimal 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 piratas 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 piratas que tenga cada uno. Esto quiere decir que por cada *tick* de reloj se ejecutará un pirata 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 pirata.

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.

Inicialmente, la primera tarea en correr es la `Idle`. Luego, en algún momento alguno de los jugadores lanzará algún pirata, 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 pirata del jugador B.

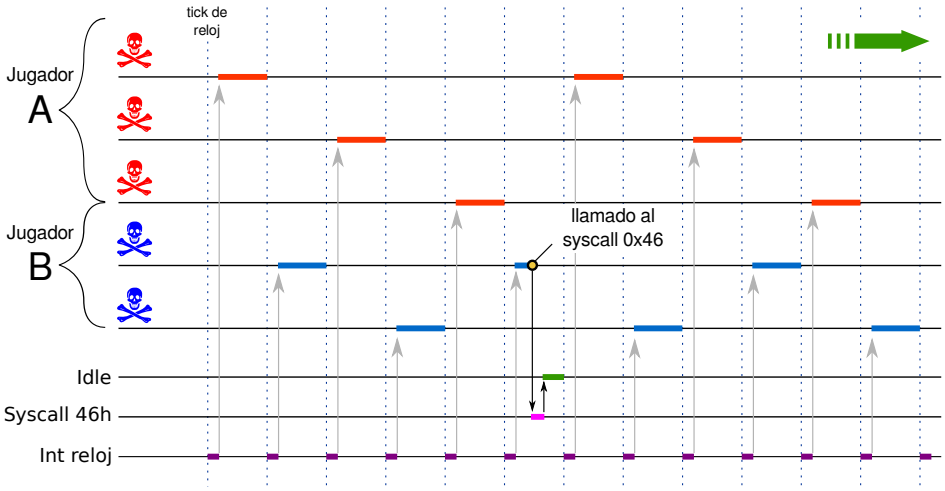

Figura 4: Ejemplo de funcionamiento del *Scheduler*

3.3. 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:

- Mapa del jugador, su puntaje, etc
- Posición de cada pirata dentro del mapa
- Pirata/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.

3.4. 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 5. 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.

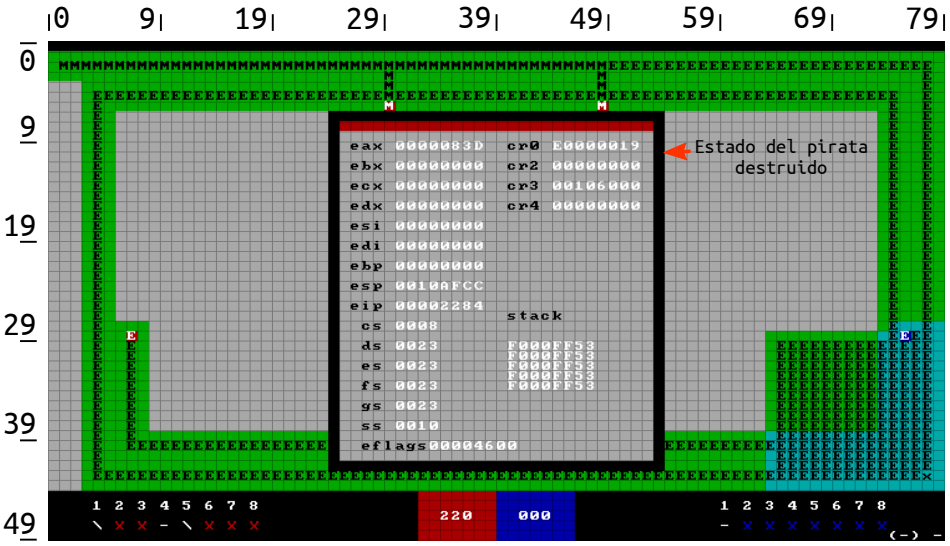

Figura 5: Pantalla de ejemplo de error

3.5. Pantalla

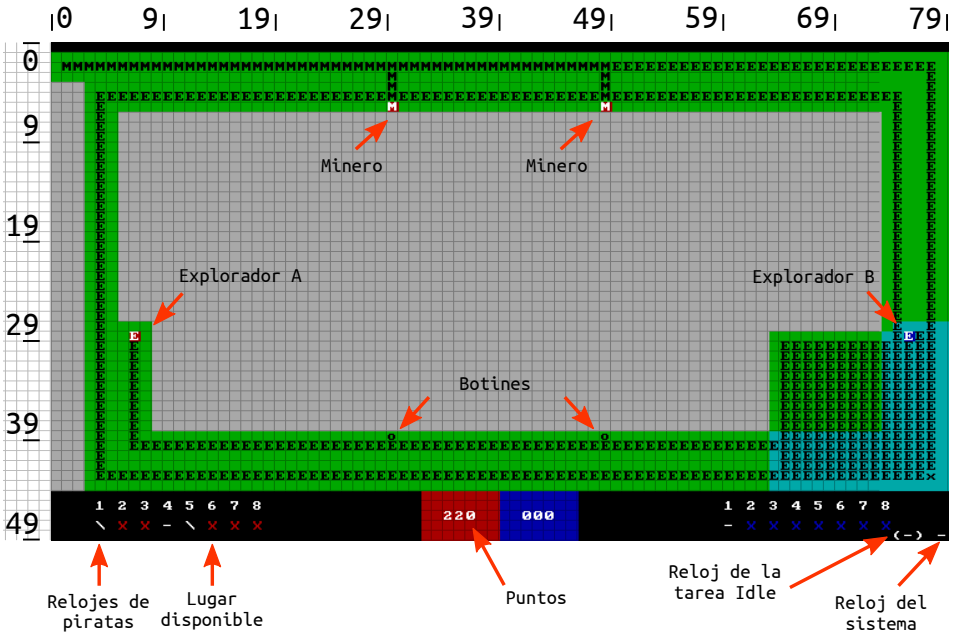

Figura 6: Pantalla de ejemplo

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

La figura 6 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. 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 posible 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.

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 500MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 7 posiciones 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) 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[1] 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 item implementar las funciones señaladas como auxiliares en `screen.h`. Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior. Es muy importante tener en cuenta que para los próximos ejercicios se accederá a la memoria de video por medio del segmento de datos de 500MB.

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.

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

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 video y pintarlo como indica la figura 6. 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 `0x003FFFFF`, como ilustra la figura 3. Además, esta función debe inicializar el directorio de páginas en la dirección `0x27000` y las tablas de páginas según muestra la figura 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 línea de la pantalla alineado a derecha.

4.4. Ejercicio 4

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

b) Escribir una rutina (`mmu_inicializar_dir_pirata`) 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 mapear dicha páginas a partir de la dirección virtual `0x0400000`. Además deberá mapear las posiciones ya descubiertas por los exploradores del jugador. Recordar que los piratas comienzan en el puerto del jugador en *el_mapa*. Sugerencia: agregar a esta función todos los parámentros que considere necesarios.

c) 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`.

d) 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. Este item no debe estar implementado en la solución final.

Nota: Por la construcción del *kernel*, las direcciones de 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 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_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.

4.6. 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_pirata`. 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 3.1.1, sin desalojar a la tarea que realiza el syscall.

h) Ejecutar una tarea `Explorador` 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` .

4.7. 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 devuelva una tarea de cada jugador por vez según se explica en la sección 3.2.

c) Crear una función `sched_tick()` que llame a `game_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_tick` por uno a `sched_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 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_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 3.4 que indicará en pantalla la razón del desalojo de una tarea.

4.8. Ejercicio 8 (optativo)

a) Crear un conjunto de 2 ~~tareas~~ piratas (explorador y minero). 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 `0x0400000`.
- Utilizar el único servicio del sistema (`0x46`).

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

b) Si consideran que sus tareas pueden hacer algo mas que completar el primer item de este ejercicio, y tienen a un audaz campión que se atreva a enfrentarse en el campo de batalla pirata, 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 pirata
   - Estrategia de exploracion y supervivencia (es decir, como se sobrevivirán en el cruel mundo pirata)

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

c) Pelicula y Video Juego favorito sobre Piratas.

5. 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 **16 de Junio** 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.