Trabajo Práctico 3
System Programming - `Pong`

Organización del Computador 2

Segundo Cuatrimestre 2019

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. Los ejercicios están inspirados en el juego ping-pong, originalmente desarrollado para la consola Magnavox Odyssey en 1972 y nombrado por Atari simplemente como Pong.

El trabajo busca construir un sistema mínimo que permita correr como máximo 12 tareas concurrentemente a nivel de usuario. Este sistema simulará un partido de tenis de mesa entre dos jugadores, `A` y `B`, sobre un tablero donde se moverán tareas (en adelante pelotas). Los jugadores se representarán como cursores ubicados en los extremos derecho e izquierdo del tablero y serán controlados con el teclado. Las pelotas serán creadas a partir de la acción de los jugadores y se moverán sobre el tablero de forma autónoma buscando acertarle al lado contrario del tablero. Cuando una pelota llega a alguno de los extremos del tablero esta puede, o bien ser interceptada por el jugador correspondiente y rebotar, o bien entrar, sumando un punto a favor del jugador del lado opuesto y destruyendo la pelota.

El movimiento de las pelotas esta dado por dos reglas: 1.- en el sentido horizontal, se mueven de forma automática, es decir, el mismo sistema es el que se encarga de moverlas cada un determinado periodo de tiempo; 2.- en el sentido vertical las pelotas son autónomas para decidir si deben moverse hacia arriba, hacia abajo o mantener la posición vertical.

Este trabajo práctico esta organizado mediante ejercicios que construyen paso a paso el sistema pedido. Los mismos proponen utilizar los distintos mecanismos que posee el procesador para programarlo desde el punto de vista de un sistema operativo.

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 además tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en *Bochs* de forma sencilla.

Al iniciar, una computadora 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 2 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_init` 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.
- `print.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 a utilizar en C para los llamados al sistema.
- `taskA1.c`, `taskA2.c`, `taskA3.c`, `taskB1.c`, `taskB2.c` y `taskB3.c` - código de las tareas (*dummy*).
- `i386.h` - funciones auxiliares para utilizar *assembly* desde C.
- `pic.c` y `pic.h` - funciones `pic_enable`, `pic_disable`, `pic_finish1` y `pic_reset`.

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. Juego

El juego simula un Pong sobre un tablero rectangular de caracteres de 78 columnas por 40 filas. El cursor de cada uno de los jugadores será dibujado en cada uno de los extremos del tablero, contando como una columna más a cada lado. La altura de cada cursor será de 7 caracteres, este se puede mover hacia arriba o abajo por medio de diferentes teclas. El movimiento de los cursores tendrá un retardo, se tomará el último movimiento dado por las teclas de cada 30 ciclos de reloj. Esto quiere decir, que a pesar de que el jugador presione la tecla muchas veces, dirección que tomará el cursor será la última presionada, es decir, el sistema almacenará la última tecla presionada y considerará su acción sobre el cursor una vez cada 30 ciclos de reloj. Cada jugador tendrá un total de 15 pelotas, de las cuales puede solamente tener 3 en juego simultáneamente. En total, el sistema debe estar preparado para ejecutar 6 pelotas simultaneas, 3 por cada jugador. Además cada jugador cuenta con tres tipos de pelotas diferentes que el jugador puede lanzar a gusto.

Las pelotas son lanzadas desde los extremos del tablero, y la altura corresponde al centro del cursor del jugador. El sistema será el encargado de ejecutar las tareas o pelotas durante un tiempo, y estas generarán las acciones correspondientes a moverse hacia arriba o abajo cada vez que el sistema consulte. El mecanismo para obtener la acción por parte del sistema será explicado en detalle más adelante. Inicialmente consiste en utilizar un servicio que permite registrar una función especial de la tarea que el sistema puede llamar para obtener la acción a realizar.

Como la pelotas no conocen su posición en el tablero, el sistema provee un servicio para consultar la posición actual. Este valor será un par de coordenadas X e Y, donde X es relativa a la distancia al jugador dueño de la tarea, e Y se corresponde con la fila dentro del tablero, contando desde arriba hacia abajo, y su valor se encontrará entre 1 y 40. Las pelotas tienen un largo recorrido hasta el otro lado de la pantalla. En este trayecto se aburren y, como son muy charlatanas, les gusta hablar con los jugadores. Para que la pelotas puedan comunicarse con el sistema, este provee un servicio para enviar mensajes e imprimirlos en pantalla.

En la pantalla se identificará además del tablero con los cursores para cada jugador, un contador de puntos y un contador de pelotas restantes para cada jugador. Además, se debe presentar en pantalla un indicador que permita saber qué pelotas están “vivas” en todo momento.

Para el control de los jugadores se dispone de las siguientes teclas:

Jugador	Tecla	Acción
Jugador A	`w`	mover hacia arriba
	`s`	mover hacia abajo
	`z`	nueva pelota tipo 1
	`x`	nueva pelota tipo 2
	`c`	nueva pelota tipo 3
Jugador B	`i`	mover hacia arriba
	`k`	mover hacia abajo
	`b`	nueva pelota tipo 1
	`n`	nueva pelota tipo 2
	`m`	nueva pelota tipo 3

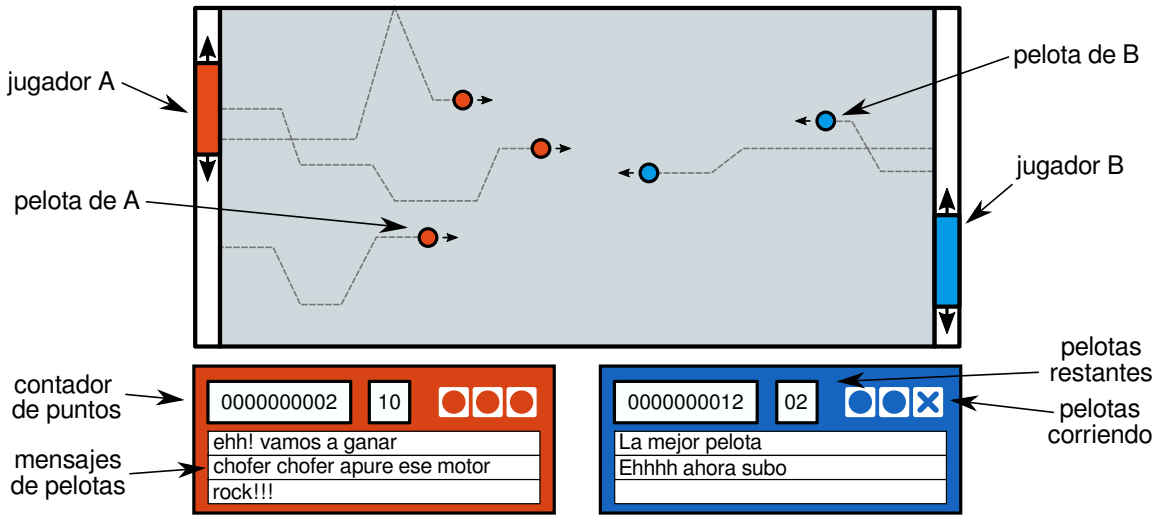

Figura 1: El Juego

3.1. Tareas

Las tareas dispondrán de cuatro servicios, `talk`, `where`, `setHandler` y `informAction` Estos se atenderán con una única interrupción (`47`). Los parámetros de cada uno se describen a continuación:

syscall	parámetros	descripción
`talk`	`in EAX=0x80000001` `in EBX=char* m`	Envia al sistema el mensaje almacenado en `m`. Este puede tener como máximo 20 caracteres.

`where`	`in EAX=0x80000002`
	`out EBX=uint32_t x`
	`out ECX=uint32_t y`
		Retorna en registros los valores `x` e `y` correspondiente a la posición actual de la pelota. El valor de `x` se cuenta desde 1 a 78, donde 1 es el extremo desde donde inicialmente fue lanzada la pelota. El valor de `y` se cuenta desde 1 a 40, donde 1 es la primera fila del tablero y 40 la última.

`setHandler`	`in EAX=0x80000003` `in EBX=f_handler_t* f`	Registra en el sistema la dirección de memoria de la función *handler* de acciones. Esta función será ejecutada por el sistema cada vez que esté requiera conocer la próxima acción que la pelota desea realizar.
`informAction`	`in EAX=0x8000FFFF` `in EBX=e_action action`	Este servicio permite retornar al sistema una vez que se este dentro de la función `handler`, ejecutando en nivel 3. Solamente será posible llamar a este servicio si se está dentro de una función *handler*. La salida del servicio será la acción (`action`) pasada como retorno en un registro.

Para cualquiera de los servicios de sistema, el registro `EAX` indica un número que permite identificar al servicio. Si este número no corresponde a ninguno de los servicios la tarea que lo llamó debe ser destruida.

El servicio `talk` como el servicio `where` puede llamarse en cualquier momento tantas veces como se desee. El resto de los registros se utilizan para los parámetros, ya sean de entrada

como de salida (indicado como `in` y `out` respectivamente). Las direcciones están codificadas respetando la siguiente enumeración:

```
typedef enum e_action {
    Up     = 1,
    Center = 2,
    Down   = 3
} e_direction_t;
```

Estas acciones corresponden a si la pelota debe moverse hacia arriba, abajo o al centro en el tablero. Existe un caso particular en el funcionamiento del tablero: cuando una pelota alcanza el borde inferior o superior del tablero entonces estas acciones deberán invertir su comportamiento. Por ejemplo, cuando una pelota llegue al extremo superior del tablero y genere la acción `Up`, la acción resultante será entonces `Down`. En la sección 3.4 se explica como se esperá que implementen este comportamiento.

Como restricción del sistema, el servicio `setHandler` podrá ser llamado una sola vez durante la ejecución de la tarea o pelota. Si el sistema recibe dos llamados al servicio, el mismo deberá matar a la tarea que representa a la pelota en cuestión.

El servicio `informAction` debe retornar el control del handler al nivel supervisor, por lo tanto el comportamiento del programa luego de llamar a este servicio no debe ser definido. Tener en cuenta que si el *handler* de una pelota agota su turno (es decir un tick de reloj) sin realizar el llamado correspondiente a `informAction`, entonces se deberá matar a esa pelota.

Cualquiera sea el caso, ninguno de los servicios debe modificar ningún registro, a excepción de los indicados anteriormente. A fin de ilustrar comportamiento de los servicios se presenta la figura 4.

3.1.1. Organización de la memoria

La memoria física estará dividida en tres áreas: *kernel*, *área libre de kernel* y *área libre de tareas*. El área de *kernel* corresponderá al primer MB de memoria, el *área libre de kernel* a los siguientes 3 MB de memoria y el área *área libre de tareas* a los siguientes 35 MB.

La administración del área libre de memoria, tanto para kernel como para las tareas, será muy básica. Se dispondrá de un par de contadores de páginas (uno para kernel y otro para tareas) los cuales serán utilizados a la hora de solicitar una nueva página. El contador correspondiente se aumentará siempre que se requiera usar una nueva página de memoria, y nunca se liberarán las páginas pedidas en ninguna de las dos áreas.

Las páginas del *área libre de kernel* serán utilizadas para datos del kernel: directorios de páginas, tablas de páginas y pilas de nivel cero. Las páginas del *área libre de tareas* serán utilizadas para almacenar los códigos, datos y pila de las tareas.

La memoria virtual de cada una de las tareas tiene mapeado el *kernel* y el *área libre de kernel* con *identity mapping* en nivel 0. Sin embargo, el *área libre de tareas* no está mapeado. Esto obligará al *kernel* a mapear el *área libre de tareas*, cada vez que quiera escribir en esta. No obstante, el *kernel* puede escribir en cualquier posición del *área libre kernel* desde cualquier tarea sin tener que mapearla.

El código, la pila y los datos de cada tarea estarán todos en una misma área compartida de 8 KB, es decir, dos paginas de memoria, para cada tarea. La copia original del código de

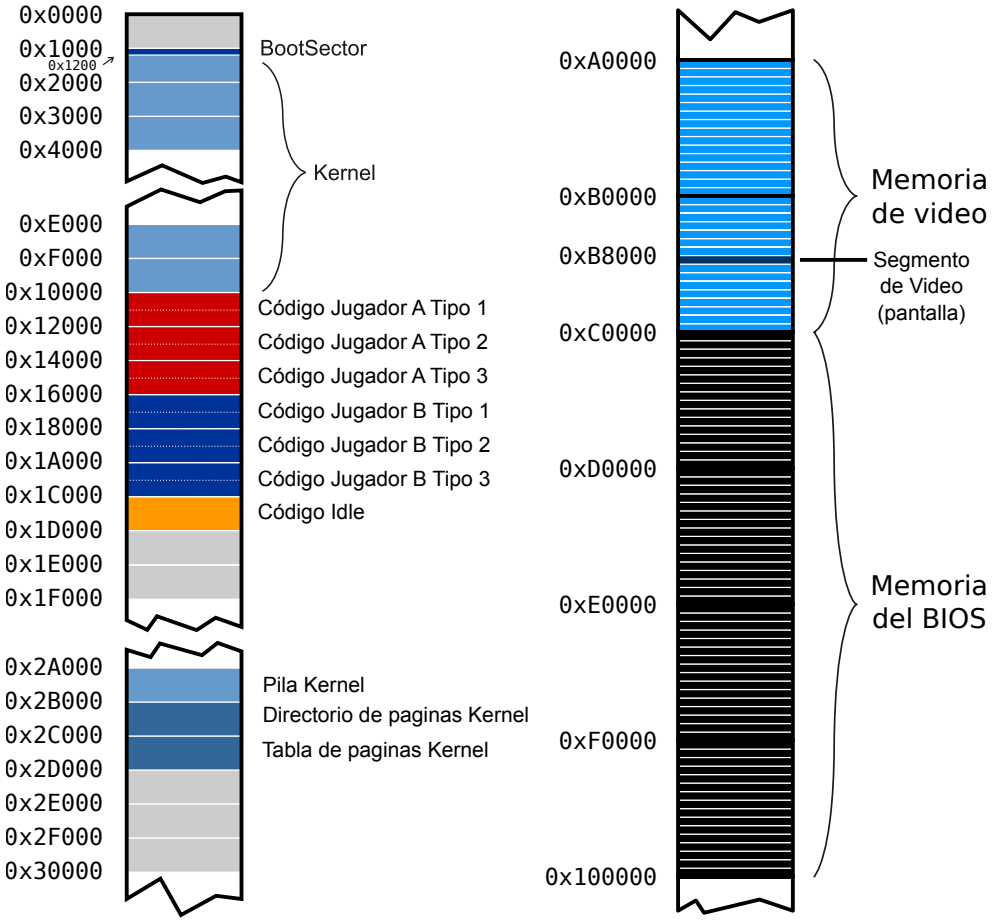

Figura 2: Mapa de la organización de la memoria física del *kernel*

cada tarea se encuentra almacenada en el kernel según indica la figura 2. El kernel tiene la copia original de las 6 tareas posibles, ya que habrán tres tipos de tareas diferentes por cada uno de los dos jugadores. Para construir una nueva tarea se deberá realizar una copia de todo el código a una nueva área dentro del *área libre de tareas*. El código de las tareas por su parte estará mapeado en nivel 3 con permisos de lectura/escritura según indica la figura 3.

3.2. Scheduler

El sistema ejecutará las tareas de forma concurrente, cada una durante un tiempo fijo denominado *quantum*. Este quantum será determinado por un *tick* de reloj. Para lograr este comportamiento se va a contar con un *scheduler* minimal que encargará de desalojar una tarea del procesador para intercambiarla por otra en intervalos regulares de tiempo.

El sistema ejecutará regularmente todas las tareas en juego cada un periodo dado por 7 tics de reloj. Durante los primeros 6 ticks de reloj, se ejecutarán las tareas, el restante será utilizado para calcular los movimientos de la tareas y dibujar la pantalla.

En cada tick de reloj el sistema tomará una tarea que aún no se haya ejecutado durante el presente periodo, y la ejecutará. Para ejecutar una tarea, dividirá el tic de reloj de la

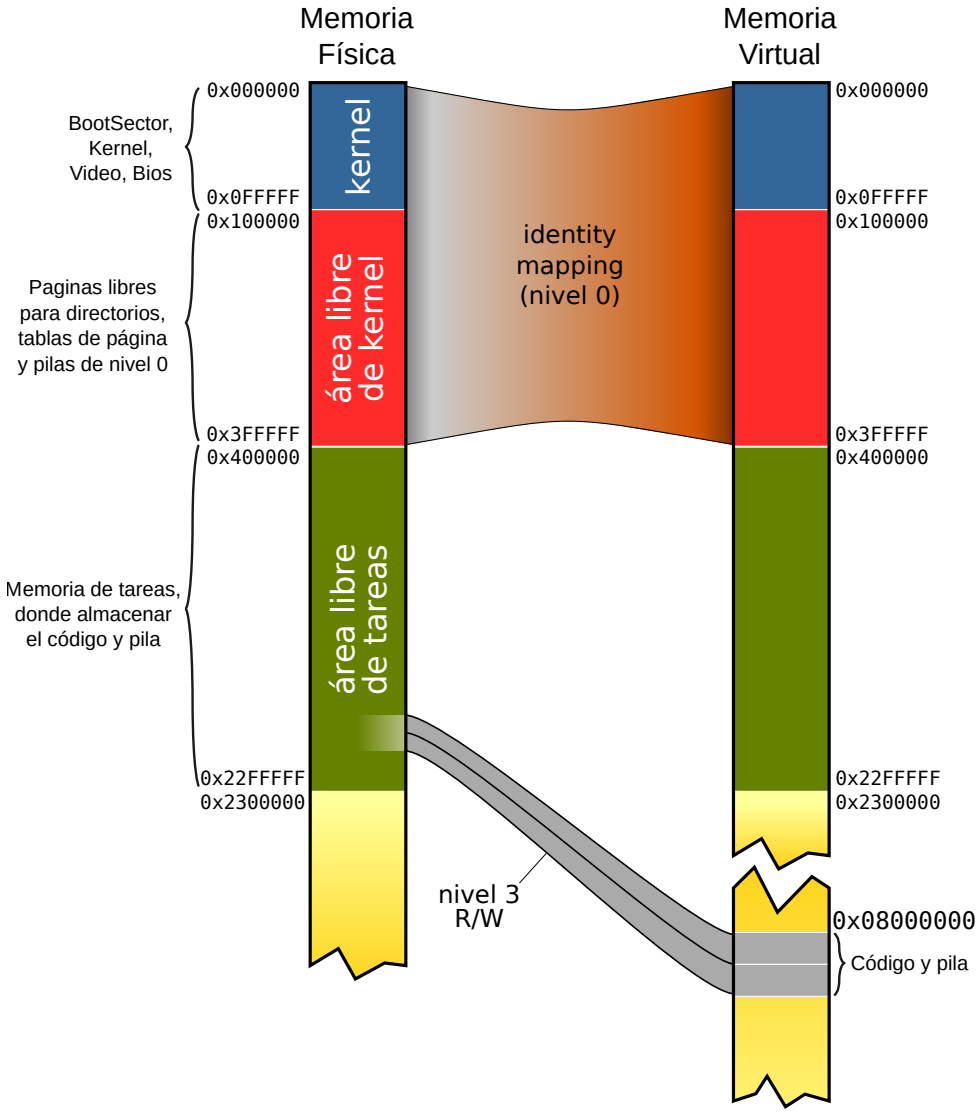

Figura 3: Mapa de memoria de la tarea

siguiente forma. Inicialmente, luego de la interrupción de reloj, se ejecutará en el contexto de ejecución de la tarea a la función *handler* de la misma. Cuando termine de ejecutar la función handler esta deberá lanzar el servicio informAction, que permite salir de la función y retornar el control al sistema. Luego, en el tiempo que resta hasta el próximo tick de reloj, se deberá ejecutar el código de la tarea en sí. Recordar, que inicialmente el *handler* de la función no esta registrado. Por lo tanto, hasta que el mismo no sea registrado, se deberá ejecutar durante todo el ciclo a la tarea, y no a la función handler. Si se da el caso en que la función handler se ejecute durante todo el tic de reloj y en ningún momento llame al servicio informAction para retornar el control al sistema, entonces pelota correspondiente deberá ser eliminada del sistema.

Cada periodo de 6 tics corresponde a ejecutar cada una de las 6 pelotas que podrían ejecutarse en todo momento en el sistema (tres de cada jugador). Si alguna de la posibles

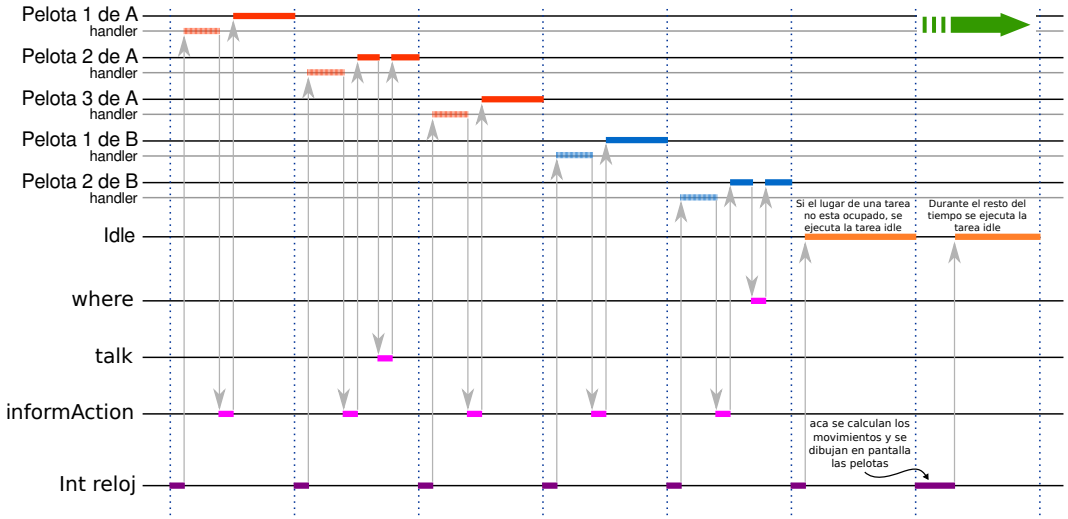

Figura 4: Ejemplo de funcionamiento del *Scheduler*

pelotas no fue lanzada aun se ejecutará la tarea `Idle` en su lugar. Una vez que las 6 tareas hayan sido ejecutadas se procederá a mover las pelotas en la pantalla y a marcar su nueva posición en el tablero. El ciclo se repetirá hasta que no queden más pelotas o tareas en juego.

Dado que los movimientos de las pelotas sólo serán considerados al final de cada periodo de 6 ticks, el orden en que realmente sean ejecutadas cada una de las tareas podrá ser cualquiera. Tener en cuenta que, a pesar de lo anterior, las tareas pueden generar problemas de cualquier tipo (ejemplo: excepciones), y por esta razón 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 ocasionada por haber llamado 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 (esto en el contexto del juego equivale a que la tarea muera, liberando el lugar para una nueva pelota).

Un punto fundamental en el diseño del *scheduler* es que deberá proveer una funcionalidad para intercambiar cualquier tarea por la tarea `Idle`. Este mecanismo será utilizado por ejemplo en el caso que una tarea genere una excepción, ya que la tarea `Idle` será la encargada de completar el *quantum* de la tarea que exploto. 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 que corresponda.

En la figura 4 se presenta un ejemplo de un ciclo completo de tareas. Notar que el jugador A tiene tres pelotas en juego, mientras que el jugador B sólamente tiene dos. Se puede observar además cómo la tareas llaman a los servicios `take` y `where`, y cómo durante cada *quantum* es ejecutada tanto la función handler de la pelota que permite obtener la próxima acción, como así también el código de la tarea en sí.

3.3. Handler de acciones

El mecanismo de funcionamiento del *handler* de acciones es simple. Inicialmente la tarea llama al servicio `setHandler` con una dirección de memoria de una función su contexto de ejecución. El sistema registra esta posición de memoria relativa al área de memoria de la tarea correspondiente. Luego, en algún otro momento, el sistema se encargará de construir un contexto de ejecución igual al de la tarea, pero modificando el inicio del programa. En este caso, utiliza como EIP la dirección de memoria previamente registrada. Este nuevo contexto de ejecución será ejecutado como una nueva tarea, que solamente va a existir a fin de ejecutar solo una vez a la función *handler*. La función *handler* será ejecutada en este nuevo contexto en nivel usuario durante un ciclo de reloj como máximo. En el caso de superar este tiempo, es decir, que ocurra una interrupción de reloj durante la ejecución de un *handler*, esta deberá ser desalojada, y la pelota correspondiente deberá ser destruída.

En caso contrario, en que el *handler* termine antes, lo hará retornando el control al sistema por medio del servicio `informAction`, el cual tiene como objetivo registrar cual es la acción que realizará la pelota durante el período actual del juego.

En la figura 5 se indican los espacios de memoria de la tarea, la primer sección corresponde a 8KB solicitados dentro del área libre de tarea. En este lugar se copiará el código de la tarea, inicialmente ubicado en el kernel. La segunda sección de la gráfica corresponde a la memoria solicitada dentro del área libre de kernel para la pila de nivel cero.

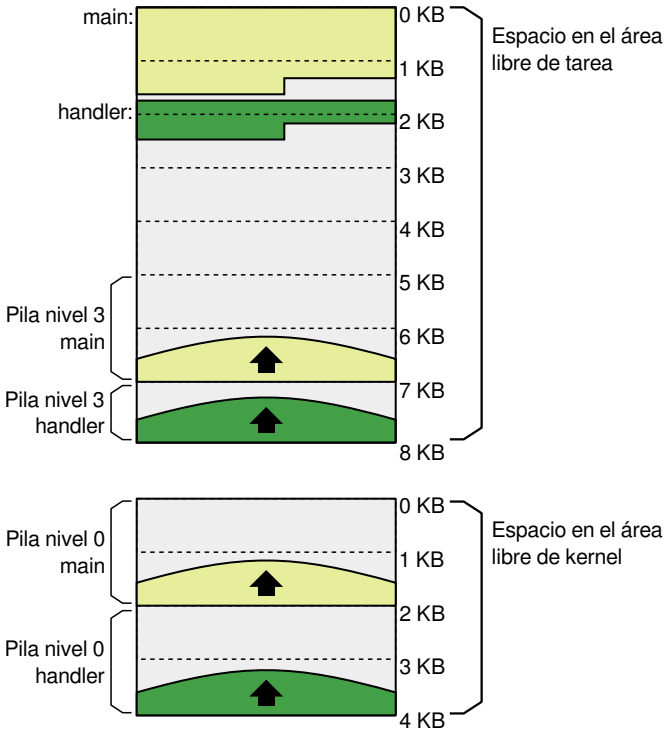

Figura 5: Mapa de la tarea

En el mismo espacio de memoria se ejecutan el *handler* y la tarea. El comienzo de la tarea se encuentra identificado en la figura como `main`. Para que puedan existir ambos contextos de ejecución, se deberá tener espacios dedicados a la pila de nivel 3 y la pila de nivel 0. Estos

espacios van a co-existir en diferentes direcciones de memoria dentro del espacio de memoria de la tarea. La figura indica cómo estos espacios serán distribuidos.

3.4. Direcciones y rebotes

Para manejar los rebotes sobre los bordes del tablero se deben considerar dos variables por tarea que almacenen el sentido en que se mueve la pelota. La figura 6 representa una versión reducida del tablero donde se ilustra el estado de estas variables a medida que la pelota recorre la pantalla y rebota contra sus bordes.

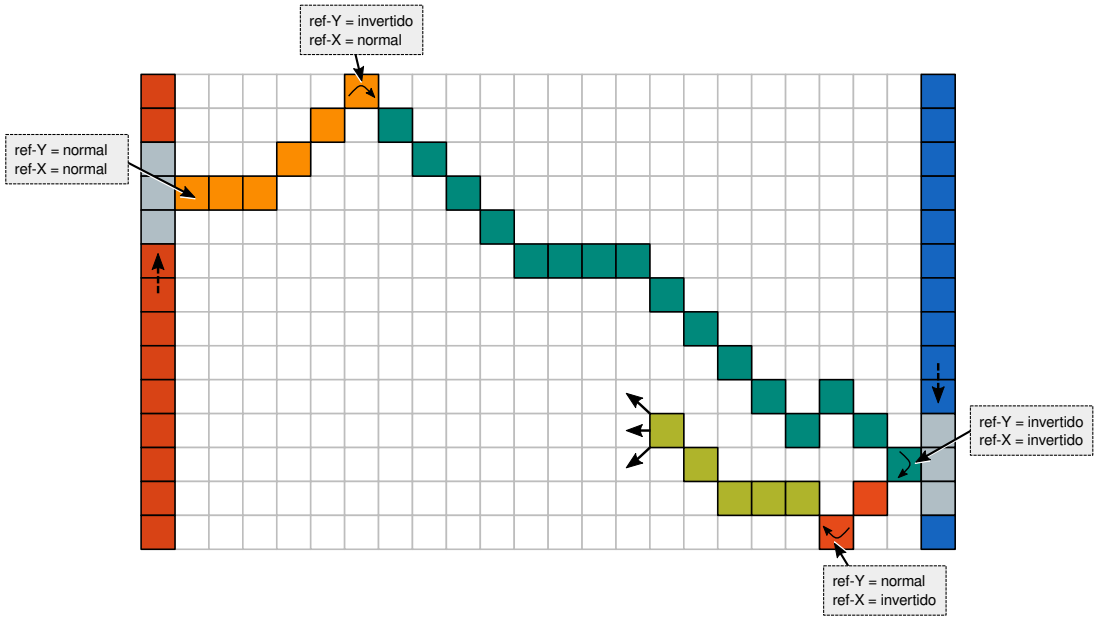

Figura 6: Direcciones y rebotes

El sentido que debe tomar la pelota debe estar controlado por el sistema. La tarea en sí misma no conoce con qué sentido se mueve, o si este fue modificado. En el sentido horizontal la pelota se va a mover por cada periodo de juego (6 ciclos) de un lado hacia el otro, rebotando en el caso que así lo dicte la lógica de juego. En el sentido vertical, podrá tener tres opciones, si el jugador opta por moverse al centro, entonces el sentido actual no influirá. Si el jugador en cambio, opta por moverse a arriba o a abajo, la interpretación de estas direcciones dependerá del sentido actual. En el caso que el sentido sea normal, la interpretación será arriba y abajo respectivamente. Si el sentido es invertido, entonces la interpretación de arriba será abajo y de abajo será arriba.

3.5. Modo debug

El sistema deberá responder a una tecla especial en el teclado, la cual activará y desactivará el modo de debugging. La tecla para tal propósito es la “y”. En este modo se deberá mostrar en pantalla la primera excepción capturada por el procesador luego de haber presionado la tecla, junto con un detalle de todo el estado del procesador, tal y como muestra la figura 7. Una vez impresa en pantalla esta excepción, el juego se detendrá hasta presionar nuevamente la tecla “y”, que mantendrá el modo debug pero borrará la información presentada en pantalla por

la excepción. La forma de detener el juego será instantánea. Al retomar el juego se esperará hasta el próximo ciclo de reloj en el que se decidirá cuál es la próxima tarea a ser ejecutada. Se recomienda hacer una copia de la pantalla antes de mostrar el cartel con la información de la tarea.

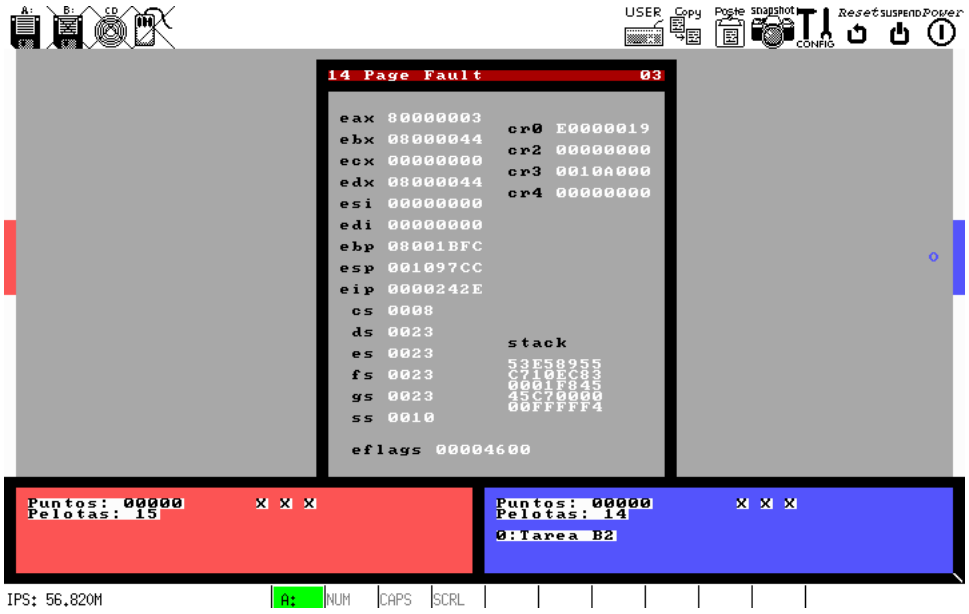

Figura 7: Pantalla de ejemplo de error

3.6. Pantalla

La pantalla presentará el tablero de 78×40. En este tablero se indicará la posición de todas las pelotas de ambos jugadores, la cantidad de puntos actuales y la cantidad de pelotas restantes. Además se deberá indicar qué pelotas o tareas están corriendo y cuáles lugares están libres para tareas pelotas.

La figura 8 muestra una imagen ejemplo de la pantalla indicando cuáles datos deben presentarse como mínimo. Se recomienda implementar funciones auxiliares que permitan imprimir datos en pantalla de forma cómoda. No es necesario respetar la forma exacta de presentar estos datos en pantalla. Se puede modificar la forma, no así los datos en cuestión.

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 163MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 13 posiciones se consideran utilizadas y por ende no deben utilizarlas. El primer índice que deben usar para declarar los segmentos, es el 14 (contando desde cero).

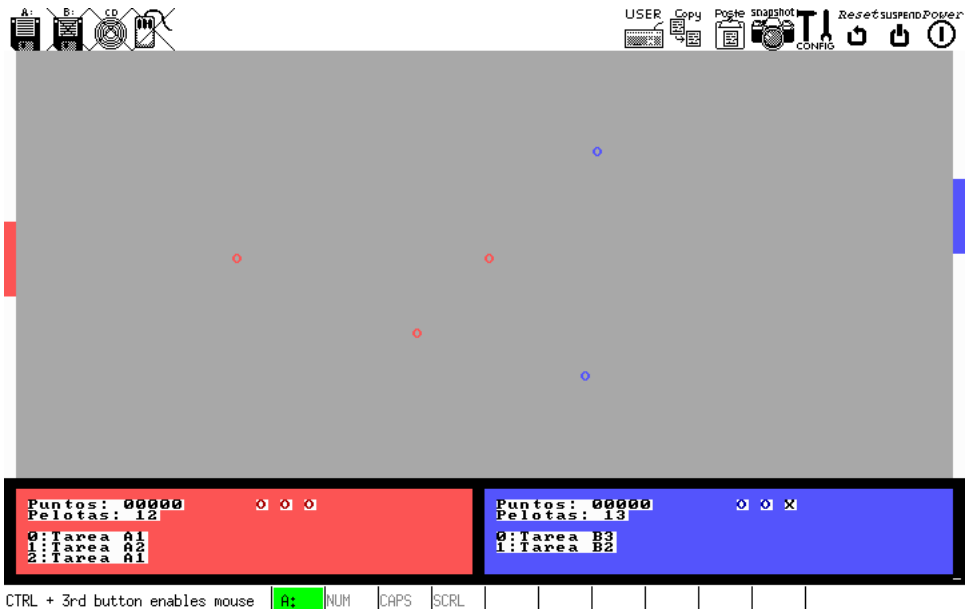

Figura 8: Pantalla de ejemplo

b) Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección `0x2B000`.

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 [1] el área del tablero con algún color de fondo, junto con las barras de los jugadores según indica la sección 3.6. 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 y no este último.

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.

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

Nota: La `IDT` es un arreglo de `idt_entry` declarado sólo 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) Completar las entradas necesarias en la `IDT` para asociar una rutina a la interrupción del reloj y otra a la interrupción de teclado. Además crear una entrada adicional para la interrupción de software `47`.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `nextClock`. 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 `nextClock` está definida en `isr.asm`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de `0` a `9`, se presente la misma en la esquina superior derecha de la pantalla.

d) Escribir la rutina asociada a la interrupción `47` para que modifique el valor de `eax` por $0x42$. Posteriormente este comportamiento va a ser modificado para atender uno de los servicios del sistema.

4.4. Ejercicio 4

a) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el *kernel* (`mmu_initKernelDir`). 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 `0x2B000` y las tablas de páginas según muestra la figura 2.

b) Completar el código necesario para activar paginación.

c) Escribir una rutina que imprima el número de libreta de todos los integrantes del grupo en la pantalla.

4.5. Ejercicio 5

a) Escribir una rutina (`mmu_init`) que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre de kernel.

b) Escribir una rutina (`mmu_initTaskDir`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3. La rutina debe mapear las paginas correspondientes a la posición indicada por el jugador dentro del mapa, a partir de la dirección virtual `0x08000000`(128MB). Luego, debe copiar el código de la tarea. Además debe mapear las paginas compartidas para el jugador en cuestión. Sugerencia: agregar a esta función todos los parámetros que considere necesarios.

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

I- `mmu_mapPage(uint32_t virtual, uint32_t cr3, uint32_t phy)`
Permite mapear la página física correspondiente a `phy` en la dirección virtual `virtual` utilizando `cr3`.

II- `mmu_unmapPage(uint32_t virtual, uint32_t cr3)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

d) A modo de prueba, construir un mapa de memoria para tareas e intercambiarlo con el del *kernel*, luego cambiar el color del fondo del primer carácter de la pantalla y volver a la normalidad. Este ítem no debe estar implementado en la solución final.

Nota: Por 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, se debe que llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

4.6. Ejercicio 6

a) Definir las entradas en la `GDT` que considere necesarias para ser usadas como descriptores de `TSS`. Mínimamente, 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 `0x0001C000`. La misma debe compartir el mismo `CR3` que el *kernel* y su pila se alojará en la misma dirección que la pila del kernel. Esta tarea ocupa 1 página de 4KB y debe ser ejecutada desde esa direccón usando *identity mapping*.

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 página de 8kb cada una según indica la figura 2. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea, la misma se ubicará según indica la figura 5. Para el mapa de memoria se debe construir uno nuevo utilizando la función `mmu_initTaskDir`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva página del área libre de kernel 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`.

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_nextTask()` que devuelve el índice en la `GDT` 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) Modificar la rutina de interrupciones `0x47` para que implemente los distintos servicios del sistema según se indica en la sección 3.1.

d) 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_nextTask()`.

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

f) Implementar el mecanismo de debugging explicado en la sección 3.5 que indicará en pantalla la razón del desalojo de una tarea.

4.8. Ejercicio 8 (optativo)

a) Crear una tarea o pelota que respete las restricciones del trabajo práctico, ya que de no hacerlo no podrán ser ejecutados en el sistema implementado por la cátedra.

   Debe cumplir:

   - No ocupar más de 8 KB (tener en cuenta la pila)
   - Tener como punto de entrada la dirección cero
   - Estar compilado para correr desde la dirección `0x08000000`
   - Utilizar los servicios del sistema correctamente

   Explicar en pocas palabras la estrategia utilizada.

b) Si consideran que su tarea pueden hacer algo más que completar el primer ítem de este ejercicio, y se atreven a enfrentarse en alocado partido de super pong, entonces pueden enviar el **binario** de sus tareas a la lista de docentes indicando el nombre de la tarea.

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

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.

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. Se espera que el informe tenga el detalle suficiente como para entender el código implementado en la solución entregada. 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 **21/11**. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia. El informe de este trabajo debe ser incluido dentro del repositorio en formato `PDF`.

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