Trabajo Práctico 3
System Programming - `Divídete y come`

Organización del Computador 2

Segundo Cuatrimestre 2018

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 como máximo 20 tareas concurrentemente a nivel de usuario. Este sistema simulará un juego entre dos jugadores, `A` y `B`, sobre un tablero donde se ubicarán las tareas. Cada tarea será capaz de leer el tablero, moverse a otra posición o duplicarse.

Sobre el tablero hay frutas de distintos sabores, al moverse una tarea a un casillero con fruta, la tarea la come y asi gana puntos. El juego consiste en mover tareas a fin obtener el mayor puntaje.

El sistema por su parte, funcionará como un arbitro del juego. Este será capaz de capturar cualquier problema que puedan generar las tareas y tomar las acciones necesarias para quitarla o reiniciarla.

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.

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 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 utilizar en C los llamados al sistema.
- `taskA.c` y `taskB.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

En el presente trabajo práctico deberemos construir un juego de dos jugadores. Cada jugador comenzará con una sola tarea en el tablero, pudiendo tener hasta un máximo de 10.

El tablero posee 50×50 celdas. Cada una de estas puede o no tener una cantidad fija de fruta, que las tareas pueden comer. Una fruta consiste en puntos, que las tareas deben capturar para ganar el juego. Cada tipo de fruta en particular representa la siguiente cantidad de puntos:

Fruta	Puntos
Uva	16
Banana	32
Frutilla	64

Por otro lado, las tareas poseen un atributo denominado *peso*. El peso limita las acciones que las tareas pueden realizar. Cuanto mayor sea el peso de una tarea, más lento se podrá mover, pero más lejos podrá mirar. A la inversa, cuanto menor sea el peso de una tarea, más rápido se podrá mover, pero no podrá mirar muy lejos en el tablero. La relación entre el peso, la distancia máxima a mirar y la distancia máxima a mover es la siguiente:

Peso	Distancia máxima a mover	Distancia máxima a mirar
64	1	64
32	2	32
16	4	16
8	8	8
4	16	4
2	32	2
1	64	1

Para moverse, las tareas indicarán la distancia en cantidad de celdas y la dirección. Si la tarea indica una distancia mayor a la permitida por su propio peso, entonces se limitará el movimiento al máximo que esta pueda moverse.

Para mirar el tablero se indicará la posición `x` e `y` relativa desde donde se encuentra la tarea. Si la suma entre `|x|` e `|y|` es mayor a la distancia máxima para mirar, entonces el resultado será nulo. La información provista al mirar el tablero está definida por `e_datatype_t`, respetando la siguiente tabla:

`Null`	La tarea no puede ver que hay.
`None`	Casilla vacía
`Player`	Al menos una tarea del jugador
`Opponent`	Al menos una tarea del oponente
`Food`	Hay fruta

El tablero por su parte es circular, esto quiere decir que el limite a derecha continua a izquierda, y el limite superior continua sobre el limite inferior y viceversa. Esta propiedad afecta tanto a la acción de mover como a la de ver el tablero.

La restante acción que pueden realizar las tareas es la de dividirse. Una vez que una tarea decide dividirse, el sistema en el siguiente turno se encargará de generar una copia de la misma, dejando como resultado dos tareas iguales (la original, y la nueva) pero con la mitad del peso para cada una. Ejemplo: si inicialmente la tarea tenia un peso de 64, al dividirse se generará una nueva tarea con peso 32 y la ya existente cambiará su peso a 32 también. Considerar que una tarea con peso 1 no puede dividirse. Además, tener en cuenta que cada jugador puede tener como máximo 10 tareas. Si ya no hay lugar disponible donde ubicar la nueva tarea la tarea original perderá la mitad de su peso, pero la copia no será creada.

El funcionamiento del juego consiste en dar a cada una de las tareas un determinado tiempo de computo donde calcular su próxima acción. Una vez que todas las tareas en el tablero calcularon su próxima acción, entonces el sistema decide pasar al siguiente turno.

Las posibles acciones a realizar por el sistema son:

- Si **una** tarea se mueve:
  - a una casilla libre → No se realiza acción.
  - a una casilla con fruta → La tarea come la fruta, el jugador suma los puntos dados por la fruta.
- Si **más de una** tarea se mueve:
  - a una casilla libre → Se suman los pesos de todas las tareas por jugador y sobreviven las tareas que tengan mayor peso total. La suma de los pesos del perdedor se adicionan como puntos al ganador. Si tienen igual peso, entonces mueren todas. Si mueren todas ninguna suma el puntaje.
  - a una casilla con fruta → Vale lo anterior y además el ganador come la fruta.

A partir de estas reglas se entiende que al final de cada turno no pueden haber dos tareas de jugadores diferentes en la misma celda. Tampoco existirán frutas en aquellas celdas que estén ocupadas por tareas.

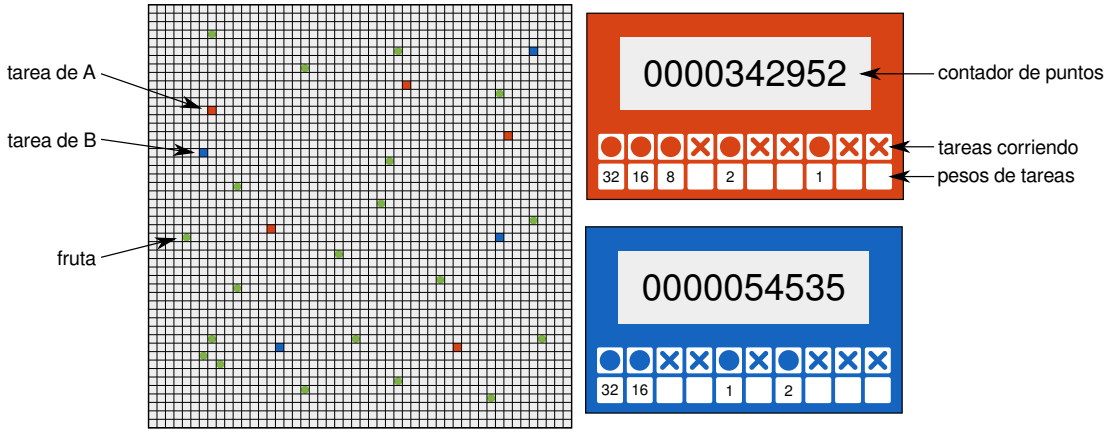

Figura 1: El Juego

3.1. Tareas

Las tareas dispondrán de tres servicios, `read`, `move` y `divide`. Estos atenderán en tres interrupciones diferentes `0x47`, `0x49` y `0x4C` respectivamente. Los parámetros de cada uno se describen a continuación:

syscall	parámetros	descripción

`read`	`in EAX=int32_t x`
	`in EBX=int32_t y`
	`out EAX=e_datatype_t r`
		Lee la posición del tablero dada por `x` e `y` (*).

`move`	`in ECX=uint32_t distance`
	`in EDX=e_direction_t dir`
	`out EAX=uint32_t max`
		Mueve la tarea la distancia dada por `distance` con la dirección `dir` (*).

`divide`	`out EAX=int32_t indicator`	Divide una tarea.

(*) Limitado según el peso de la tarea que llame al servicio.

Para el servicio `read`, cualquier parámetro para `x` e `y` es válido, ya que este será limitado por la máxima visión que pueda alcanzar la tarea como fue descripto anteriormente. El valor de retorno estará dado por `EAX`, respetando la codificación: `Null=0`, `None=10`, `Player=20`, `Opponent=30` y `Food=40`.

Los parámetros de `move`, por otro lado, deben respetar los siguientes valores para codificar direcciones: `Right=1`, `Left=2`, `Up=3` y `Down=4`. El valor de retorno también estará dado por `EAX`, siendo este la distancia efectiva en que la tarea se logro mover.

El servicio `divide` no toma parámetros. Este crea una copia de la tarea original, y divide el peso de la misma entre esta y la copia. Ambas tareas estarán ubicadas inicialmente en la misma celda. La única diferencia entre ambas será el valor de retorno del servicio. Si el valor es `0`, entonces se trata de la tarea original. Si el valor es `1`, la tarea es la copia. Existe el caso, en que todos los lugares donde ubicar nuevas tareas estén ocupados (se pueden crear como máximo 10 tareas por jugador), en este caso, la nueva tarea no será creada y el servicio retornará el valor `-1`.

Por último, el comportamiento de los servicios una vez ejecutados es diferente en cada caso.

- Para el servicio `move` o `divide`, se debe comenzar a ejecutar la tarea `Idle` durante el resto del *quantum*. Este último comportamiento es ilustrado por la figura 4.
- El servicio `read` por su parte tiene dos comportamientos diferentes. Si se llama al servicio *a lo sumo* una cantidad dada de veces, entonces se retorna el control a la tarea. Si la cantidad de veces que se llamo al servicio supera un limite dado, se debe comenzar a ejecutar la tarea `Idle` durante el resto del *quantum*. El limite de veces que se puede llamar al servicio será igual al peso de esa tarea, es decir, una tarea de peso 64 puede llamar a `read` y retomar su ejecución 64 veces en un mismo turno, pero será reemplazada por la tarea `Idle` en la llamada número 65.

3.1.1. Lógica de juego

Inicialmente se cuenta con dos tareas de peso 64 para cada jugador. Estas comenzarán a ejecutar dando un ciclo del scheduler para cada una. Esto quiere decir que cada tarea ejecutará hasta que llegue una interrupción de reloj, momento en que será desalojada para ejecutar la siguiente.

Una vez que todas las tareas en el tablero ejecutaron una vez, se dice que pasó un turno. Entonces es momento en que el sistema determine que pasará con cada tarea: si estas deben ser destruidas, o deben comer y sumar puntos, o simplemente no sucede nada. Cualquiera sea el caso, el sistema será el encargado de tomar estas determinaciones sobre las tareas. Luego, una vez resueltos todos los conflictos, se dará comienzo al próximo turno, ejecutando nuevamente todas las tareas. Esta lógica se repetirá de forma cíclica, de forma permanente mientras el juego esté corriendo.

La tareas son programas ejecutados concurrentemente. Como tales, estos pueden producir errores en cualquier momento de su ejecución, y de ser así las tareas correspondientes deben ser desalojadas inmediatamente para nunca mas volver a correr, dejando libre el espacio que estaba ocupando. Esto significa que cuando una tarea muere, libera un lugar para una nueva tarea.

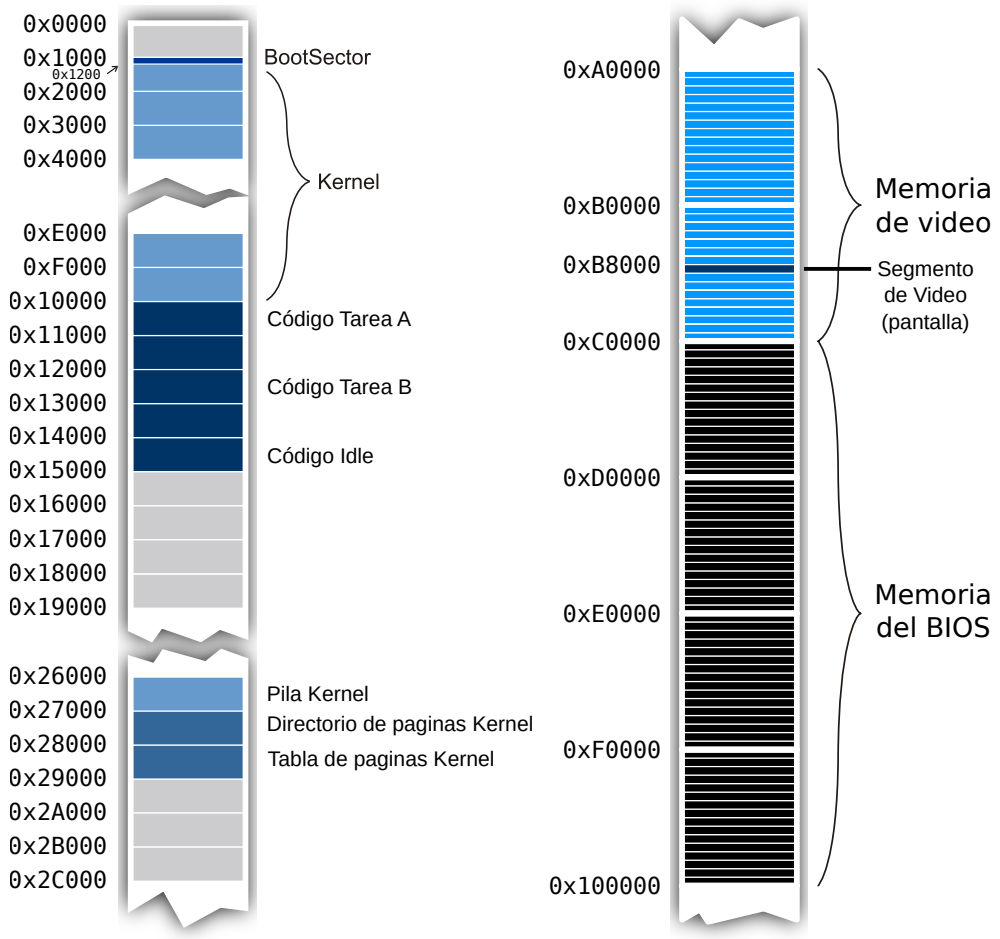

Figura 2: Mapa de la organización de la memoria física del *kernel*

El juego termina en dos casos: cuando ya no hay más fruta que comer, o bien cuando todas las tareas de un jugador están muertas (caso en que ya no le sería posible crear nuevas). El ganador en ambos casos será el que haya obtenido mayor cantidad de puntos totales.

3.1.2. Organización de la memoria

La memoria física estará dividida en tres áreas: *kernel*, *libre de kernel* y *libre de tareas*. El área *kernel* corresponderá al primer MB de memoria, el área *libre de kernel* a los siguientes 3 MB de memoria y el área *libre de tareas* a los siguientes 4 MB.

La administración de las áreas libres de memoria será muy básica. Se tendrá un contador de paginas por cada una de estas áreas, a partir del cual se solicitará una nueva página. Este contador se aumentará siempre que se requiera usar una nueva página de memoira, y nunca se liberarán las páginas pedidas.

Las paginas del área *libre de kernel* serán utilizadas para datos del kernel: directorios de paginas, tablas de paginas y pilas de nivel cero. Las paginas del área *libre de tareas* serán utilizadas para datos de las tareas, paginas de código, datos y pila de las tareas.

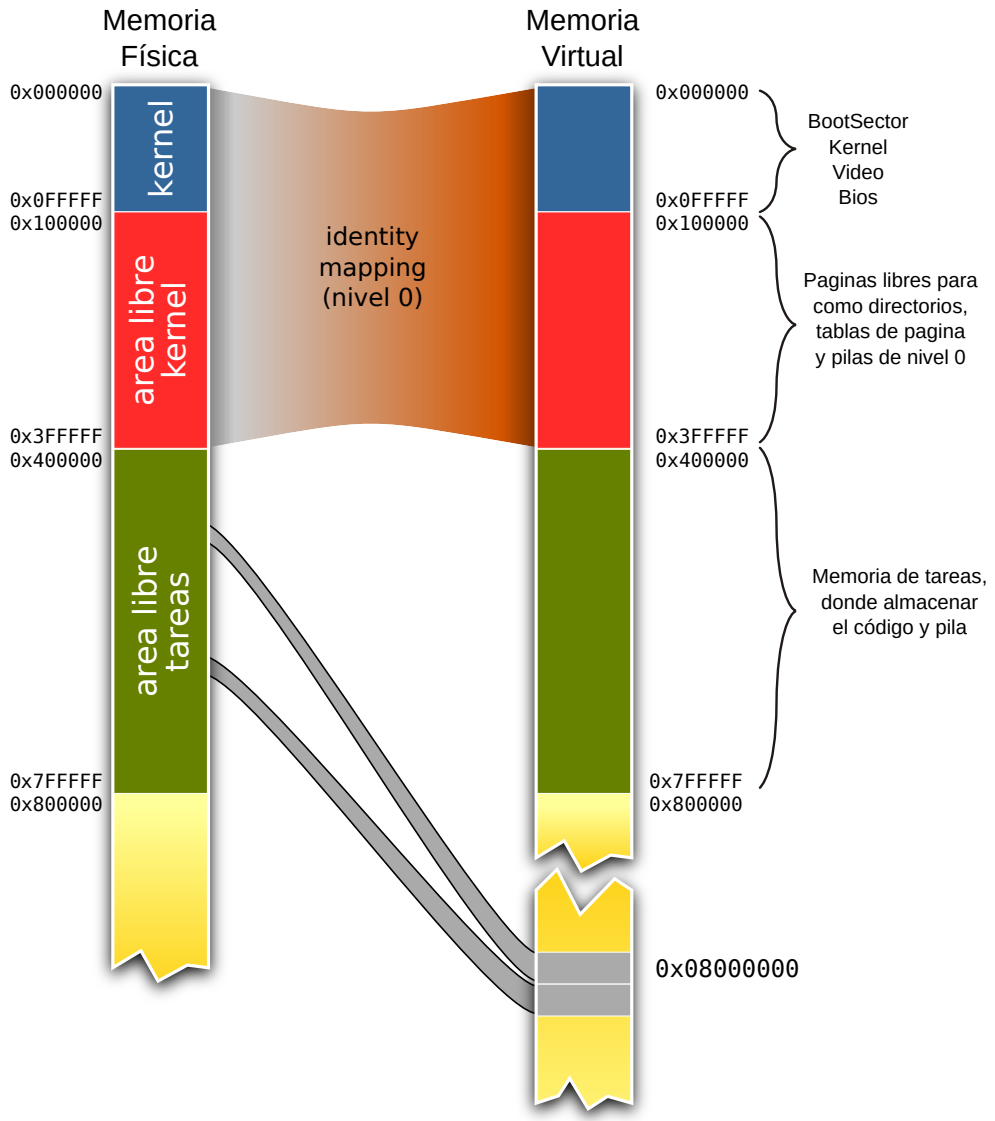

Figura 3: Mapa de memoria de la tarea

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

El código, pila y datos de las tareas estará compartido en un área de 8 KB. La copia original del mismo se encuentra almacenada en el kernel. Para construir una nueva tarea se debe realizar una copia.

El código de las tareas estará mapeado en nivel 3 con permisos de lectura/escritura según indica la figura 3.

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 en este caso de 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.

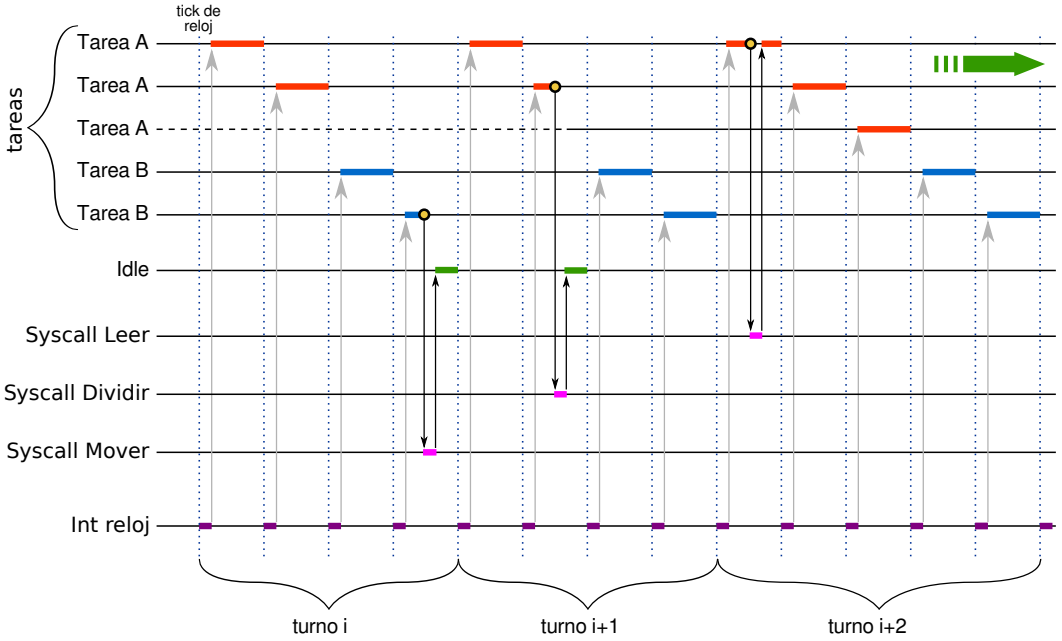

Figura 4: Ejemplo de funcionamiento del *Scheduler*.

Si bien se cuenta con tareas de dos jugadores, las tareas serán ejecutadas de forma indistinguible por el sistema. Esto quiere decir que no importa el orden en que las tareas son ejecutadas, ya que los cambios en el juego se verán luego de que todas las tareas ejecutaron su parte del turno. Considerar además que las tareas creadas por el servicio dividir no podrán volver a ejecutarse hasta el siguiente turno al que hayan elegido dividirse.

Las tareas pueden generar problemas de cualquier tipo, 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 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 (esto en el contexto del juego equivale a que la tarea muera, liberando el lugar para una nueva tarea).

Un punto fundamental en el diseño del *scheduler* es que debe proveer una funcionalidad para intercambiar cualquier tarea por la tarea `Idle`. Este mecanismo será utilizado al momento de llamar a servicios 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 que corresponda.

3.3. 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 propósito 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 instantáneamente, al retomar el juego se esperará hasta el próximo ciclo de reloj en el que se decida 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.

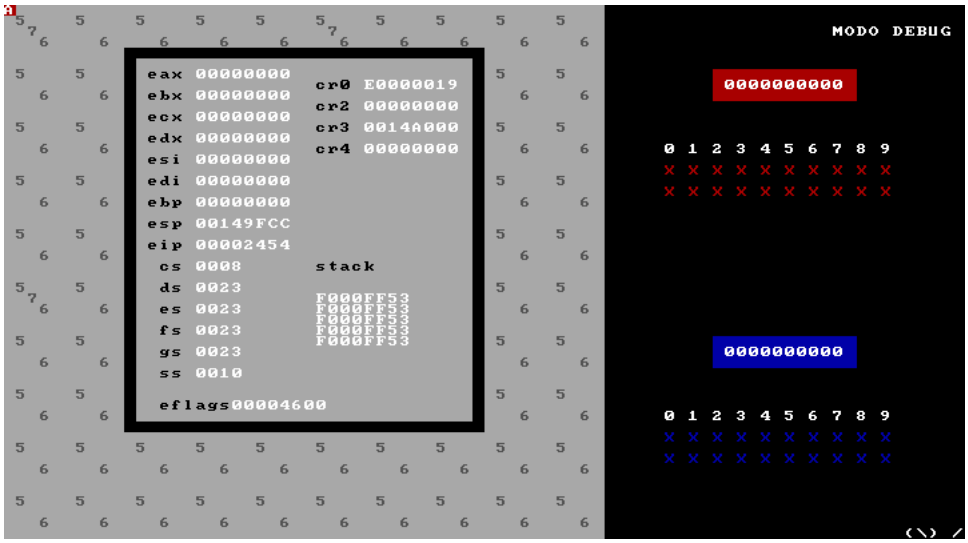

Figura 5: Pantalla de ejemplo de error

3.4. Pantalla

La pantalla presentará el tablero de 50×50. En este mapa se indicará la posición de todas las tareas de ambos jugadores y la posición de la fruta.

La figura 6 muestra una imagen ejemplo de 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.

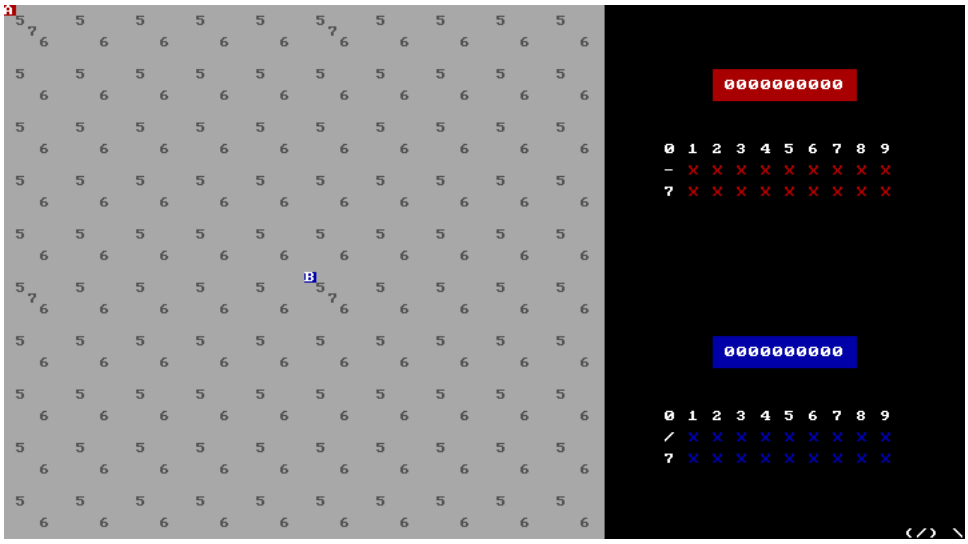

Figura 6: Pantalla de ejemplo

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 666MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 21 posiciones se consideran utilizadas y no deben utilizarse. El primer índice que deben usar para declarar los segmentos, es el 22 (contando desde cero).

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

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

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar [1] el área del tablero con algún color de fondo, junto con las barras de los jugadores según indica la sección 3.4. 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 vídeo 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 pro-

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

blema se produjo e interrumpir la ejecución. Posteriormente se modificarán estas rutinas para que se continúe la ejecución, resolviendo el problema y desalojando a la tarea que lo produjo.

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

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

4.3. Ejercicio 3

a) 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 tres entradas adicionales para interrupciones de software, `0x47`, `0x49` y `0x4C`.

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 `0x47` 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 `0x27000` 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 y en el área libre de tareas (dos contadores de paginas libres).

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 solicitar dos

páginas del area libre de tareas donde copiar el código de la tarea y mapear dicha página a partir de la dirección virtual `0x08000000`(128MB). 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) 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 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`. 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 `0x00014000`. 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 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 crecerá desde la base de la tarea. 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 pagina 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 las rutinas de la interrupciones `0x47`, `0x49` y `0x4C` para que implementen 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 3.3 que indicará en pantalla la razón del desalojo de una tarea.

4.8. Ejercicio 8 (optativo)

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

   Deben 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 qué 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 a batalla por alimentarse de mucha fruta, 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. 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 **13/11**. 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.

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