DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Segundo Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Mamani Aleman, J. Martin	630/17	mr.tinchazo@gmail.com
Chami, Uriel Alberto	157/17	uriel.chami@gmail.com
Gonzalez Omahen, Augusto E.	496/17	gonzalezomahen@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Este trabajo práctico consiste en comprender y programar las partes esenciales de un kernel de manera que sea posible ejecutar una versión minimalista del juego Pong. El objetivo de este trabajo es comprender los fundamentos básicos de un sistema operativo y llevar a la práctica conceptos como segmentación, paginación, memoria, interrupciones, tareas, scheduler y protección.

Para el desarrollo de las distintas secciones del trabajo, utilizamos los lenguajes C y ASM según la conveniencia en cada caso. Sin embargo, para poder empezar a escribir código, la cátedra nos proporcionó un conjunto de archivos y nos comprometimos a comprender su funcionamiento. De estos archivos solamente hemos modificado los siguientes:

- defines.h
- game.c
- game.h
- gdt.c
- gdt.h
- idt.c
- idt.h
- isr.asm
- isr.h
- kernel.asm
- mmu.c
- mmu.h
- sched.c
- sched.h
- screen.c
- screen.h
- tss.c
- tss.h

Los demás archivos originales no han sufrido modificaciones.

2. Estructuras y variables propias

Además, creamos el archivo gameStructs.h, que contiene todas las estructuras y variables que creamos para poder manejar la información del juego. Allí se encuentran:

- **quantum** Una variable que indica justamente qué *quantum* tiene lugar actualmente en el scheduler (y de esta forma indica qué tarea se está ejecutando). Siempre usaremos esta variable tomándole módulo 7.

- **handlers_activos[6]** Un arreglo de 6 enteros de 32 bits. Las primeras 3 posiciones corresponden en orden a las pelotas 1, 2 y 3 del jugador A y las restantes 3 a las del jugador B. Esta característica se repetirá de aquí en adelante para todos los arreglos de 6 posiciones. Si *handlers_activos[0]* es igual a cero, entonces el handler de la pelota 1 del jugador A está inactivo. En caso contrario, esta entrada del arreglo contiene un puntero al handler de esta tarea.

- **ejecutando_handler[6]** Un arreglo de 6 enteros utilizados como variable booleana. Cada posición se corresponde a una tarea de un jugador como en *handlers_activos*. Si *ejecutando_handler[6]* es uno, entonces actualmente se está ejecutando el handler de la tarea 3 del jugador B. En caso contrario, no se está ejecutando esta.

- **pelotas_vivas[6]** Otro arreglo de 6 enteros usados como booleanos. Si *pelotas_vivas[1]* es 1, entonces la pelota en el slot 2 del jugador A está viva.

- **coordenadaPelota** Una estructura que guarda los campos X e Y que son las coordenadas de una pelota en el tablero. Además guarda la variable direccionX, que vale 0 si la pelota se mueve hacia la izquierda y 1 en caso contrario, y la variable direccionY que vale 0 si no chocó con el techo aún (le llamamos no invertido) y 1 en caso contrario (invertido).

- **e_action_t** Estructura dada por la cátedra. La hemos movido a este archivo.

- **mensajesJugadorA[3][21]** y **mensajesJugadorB[3][21]** Ambos son arreglos de 3 strings (es decir, arreglos de char) donde cada uno es de longitud máxima 20 (y una posición más para el caracter nulo).

- **coordsPelotasPorSlot[6]** Un arreglo de 6 estructuras de tipo *coordenadaPelota*. Como su nombre lo dice, cada posición guarda las coordenadas de la tarea correspondiente.

- **alturaJugadorA** y **alturaJugadorB** Enteros de 32 bits que indican en qué fila se encuentran los jugadores (recordar que siempre se mueven en la misma columna, por lo que basta guardar su coordenada en Y).

- **movimientosPendientesPorSlot[6]** Un arreglo de 6 estrcturas de tipo e_action_t. Cada posición contiene el próximo movimiento a dibujar de la tarea correspondiente en la pantalla. Esto se hará cuando llegue el turno del séptimo ciclo de clock, y es por esto que debemos guardar esta información hasta llegar al séptimo **quantum**.

- **kernelLevelTasksStacks[6]** Un arreglo de 6 punteros. Cada tarea debe tener su stack de nivel 0 almacenado en el área libre del kernel, por lo que en cada posición se encuentra su esp0 correspondiente.

- **userLevelTasksCodeAndStacks[6]** Un arreglo de 6 enteros de 32 bits. Análogamente, cada tarea tiene su propio **cr3** en el área libre del kernel y entonces cada posición guarda el **cr3** de su tarea correspondiente.

- **pelotasDisponiblesA** y **pelotasDisponiblesB** Variables que informan cuántas pelotas le quedan a cada jugador. Inicialmente valen 15.

- **puntajeA** y **puntajeB** Variables que informan el puntaje actual de cada jugador. Hemos decidido que cada vez que el adversario no logre devolver una pelota, la recompensa será de 100 puntos para el otro jugador.

- **modoDebug** Un entero utilizado como variable booleana. Si vale uno, entonces el modo debug está activado. En caso contrario, está desactivado y el juego corre normalmente, es decir que al presentarse una excepción no se detendrá. Matará a la tarea y continuará la ejecución.

- **enPausa** Es una variable booleana, si está en 1, significa que el modo debug se encontraba prendido y se dio una excepción. Esta variable nos permite detener el dibujado de la pantalla y la ejecución de tareas hasta que el jugador aprete “y“

- **exception_msg[40]** Un arreglo de 40 caracteres utilizado para poder imprimir e informar la excepción generada en el modo debug.

3. Explicación de los ejercicios

A continuación explicaremos cómo resolvimos el trabajo práctico y los motivos de cada una de nuestras decisiones tomadas. Para esto, la explicación seguirá el orden propuesto por la cátedra: cada uno de los 7 ejercicios tendrá su propia sección y allí se encontrarán las explicaciones y decisiones correspondientes a dicho punto.

3.1. Ejercicio 1

El objetivo era completar la tabla de descriptores globales de modo tal que los datos y los códigos de nivel 0 y 3 tengan cada uno su segmento pertinente. Estos segmentos debían direccionar los primeros 163MB de memoria. Es por esto que el límite de los cuatro será este mismo valor expresado en hexadecimal como 0xA2FF. Este número es (163MB - 0xFFF) * 4K, es decir que 0xA2FF $<<$ 12 + 0xFFF = 163MB - 1 dado que la granularidad está activada (atributo G=1).

Además, la base en todos los segmentos es 0. En la **gdt**, por restricción del trabajo práctico, las primeras 13 posiciones se consideran reservadas y por ende no deben ser utilizarlas. Es por esto que el primer segmento que definimos ocupa la posición 14.

En el archivo defines.h definimos 4 macros para los índices de cada uno de estos segmentos en la **gdt**. Los índices van de 14 a 17 y se explicarán en conjunto los atributos de ambos segmentos de código ya que comparten la gran mayoría de los campos salvo los indicados explícitamente. Lo mismo haremos con los segmentos de datos.

Atributos de la GDT de los segmentos de Código:

límite: 0x0A2FF

base: 0x00000000

s: 1

dpl: Será 3 para el segmento de código de usuario y 0 para el de kernel

type: Este atributo define el tipo de segmento para los chequeos de protección general, en este caso decidimos que sea código execute only, es decir: 1000

p: Bit *present* = 1 porque el segmento siempre estará presente en memoria.

avl: Bits en 0 (son reservados de intel)

l: Bit en 0 porque estamos en 32 bits

db: Bit en 1 porque estamos en 32 bits

g: Bit de granularidad en 1 porque contamos de a 4kb el límite. Es decir, limiteFinal = limite << 12 + 0xFFF

Atributos de la GDT de los segmentos de Datos:

límite: 0x0A2FF

base: 0x00000000

s: 1

dpl: Será 3 para el segmento de datos de usuario y 0 para el de kernel

type: Este atributo define el tipo de segmento para los chequeos de protección general, en este caso decidimos que sea data read/write, es decir: 0010

p: *present* = 1 porque el segmento siempre estará presente en RAM.

avl: 0 (bits reservados de intel)

l: Bit en 0 porque estamos en 32 bits

db: Bit en 1 porque estamos en 32 bits

g: Bit de granularidad en 1 porque contamos de a 4kb el límite. limiteFinal = limite << 12 + 0xFFF

Para habilitar el bit 20, es necesario activar la compuerta A20. Esto se logra llamando a la función A20_enable, proporcionada por la cátedra.

Después se carga la **gdt** mediante la instrucción *LGDT*, que modifica el registro *GDTR* que justamente es un puntero a la tabla GDT.

Luego seteamos el bit PE (Protection Enable) del registro **cr0** que permite pasar a modo protegido. Como este bit es el menos significativo, basta tomar el valor de **cr0** y modificarlo a 1 mediante una operación 'OR' entre **cr0** y **1**. Pero esto no se puede hacer de manera directa con lo cual primero cargamos **cr0** en **eax**, efectuamos **OR EAX, 1** y finalmente movemos el valor de **eax** a **cr0**.

Ahora sí, pasamos a modo protegido mediante un salto de tipo "far"mediante la instrucción *farJump:* **jmp 0x0070:modoProtegido**.

El selector de segmento es 0x0070 (en binario $000000001110|0|00$). Esto significa que RPL=00, TI=0, índice = 14 = 00001110 (y se lo extiende con 5 ceros en la parte mas significativa). Podíamos expresar

este selector como 14<<3, forma que de ahora en adelante usaremos.
A partir de la etiqueta modoProtegido operamos en 32 bits para que el compilador de ASM no genere fallos, por lo que aclaramos que a partir de esta línea de código las instrucciones se refieren a dicha arquitectura mediante una directiva de compilador que es BITS 32.

Ahora, para establecer los selectores de segmentos de DS (data segment) y SS (stack segment) usamos *ax*, que valdrá 16 shifteado a la izquierda 3 bits. Con esto, estamos indicando que se usa el segmento de la posición 16 de la **gdt**, es decir, el segmento de datos de nivel cero.

Para el segmento ES (segmento extra utilizado para la pantalla solo en el ejercicio 1) utilizamos el índice 18, es decir, GDT_PANTALLA_KERNEL en defines.h. Este segmento está compuesto por los siguientes campos:

Límite: 1 pues con la granularidad en 1 tendremos (1<<12)+0xFFF bytes dedicados a la pantalla

base: 0x000B8000 por consigna

type: data read/write = 0010

s: 1

dpl: 0 pues la pantalla la maneja el Kernel

p: 1

avl: 0

l: 0 porque estamos en 32 bits

db: 1

g: 1

Para establecer la base de la pila, basta con asignarle a ESP el valor 0x0002B000 pues la consigna nos indica que la pila comienza en esa dirección.

Para imprimir un mensaje de bienvenida, utilizamos la macro
"PRINT_TEXT_PM START_PM_MSG, START_PM_LEN, 0X07, 0, 0".
Los parámetros son un puntero al mensaje (start_pm_msg), la longitud del mensaje (start_pm_len), el color (0x07, es decir, C_BG_BLACK con C_FG_LIGHT_GREY) y la fila y columna (0,0)

Para limpiar la pantalla, utilizamos a **eax** como contador de bytes, que es el doble que la cantidad de píxeles ya que la pantalla es de 80x50, hay 4000 píxeles y cada píxel ocupa 2 bytes. El ciclo se basa en escribir en el segmento ES (más un offset dado por eax) el valor CARACTER_LIMPITO definidio al principio de *kernel.asm* e incrementar de 2 en 2 **eax**.

Después de esto, imprimimos el tablero y las áreas donde va la información pertinente a cada jugador con screen_drawBox.

3.2. Ejercicio 2

Para poder manejar interrupciones, nos encargamos de inicializar la tabla IDT llamando desde el archivo *kernel.asm* a la función *idt_inicializar*. Aquí tenemos una variable global en *idt.c* llamada *idt*, que es un arreglo de valores de tipo *idt_entry*.

Las primeras 32 entradas de la IDT corresponden a interrupciones de tipo excepción. Las inicializamos mediante la macro *IDT_ENTRY*, que se encarga de modificar los campos en la IDT. Cabe resaltar que al ser excepciones, las 32 entradas tendrán como atributos 0x8E00. Esto es en binario $1|00|01110|000|00000$ donde los bits 13 y 14 representan el nivel de privilegio **DPL** = 00.

Asimismo, cada una de estas excepciones posee una rutina que indica en pantalla qué problema se produjo e interrumpe la ejecución. Esto se realiza mediante la macro *ISR* (definida en el archivo *isr.asm*) que llama a la función *print_exception* (definida en el archivo *screen.c*) y por ahora haltea la ejecución mediante la instrucción *jmp $*. Más adelante usaremos estas rutinas para desalojar tareas que generen excepciones.

Además, hemos generado una excepción para probar esta sección del trabajo práctico. El código, que pretendía dividir por cero y ahora se encuentra comentado, es el siguiente:

mov eax, 0

div eax

Al intentar efectuar esta operación, pudimos corroborar que se imprimía por pantalla el mensaje *"Divide-by-zero Error"*.

3.3. Ejercicio 3

La entrada 32 de la IDT corresponde a la interrupción del clock, cuya rutina de atención está dada por la función *_isr32* (definida en *isr.asm*). Por ahora, esta rutina consiste en llamar a la función nextClock, que imprime un simpático reloj en una esquina de la pantalla, y llamar a la función *pic_finish1*, que informa que la interrupción por hardware ha sido atendida. Más adelante modificaremos esta rutina de atencion de la interrupción 32 y explicaremos su funcionamiento.

Por otra parte, la entrada 33 de la IDT corresponde a la interrupción dada por el teclado. Su rutina está dada por la función *_isr33* en donde preservamos todos los registros mediante la instrucción pushad, leemos la tecla que ha sido presionada y la guardamos en el registro al mediante la instrucción *in al, 0x60*. Acto seguido, llamamos a la función *print_number* que recibe como parámmetro la tecla presionada. Si dicha tecla era uno de los números (de 0 a 9), entonces se imprime este mismo número en la esquina superior derecha. Luego, *_isr33* finaliza llamando a la función *pic_finish1* y reestableciendo los valores de todos los registros guardados mediante la instrucción popad.

Finalmente, la entrada 47 de la IDT corresponde a una interrupción realizada por el usuario, por lo que a esta entrada la inicializamos aparte mediante la macro *IDT_ENTRY_DPL3* que funciona como *IDT_ENTRY* salvo que los atributos ahora son 0xEE00. En binario, esto es $1|11|01110|000|00000$. Como podemos ver, ahora los bits 13 y 14 indican nivel de privilegio 3 (**DPL** = 11) pues esta interrupción la produce el usuario, no el Kernel. Inicialmente, toda la rutina consiste en la instrucción *mov eax, 0x42*. Luego será modificada.

Volviendo al archivo *kernel. asm*, cargamos la IDT mediante la instrucción *lidt* [IDT_DESC] ya que IDT_DESC es de tipo *idt_descriptor*. Esta es una estructura que contiene el tamaño de la idt y su dirección, por lo que es suficiente para cargarla.

Luego, configuramos el controlador de interrupciones mediante una serie de instrucciones:

1. call pic_reset (función dada por la cátedra).
2. call pic_enable (función dada por la cátedra).
3. sti: vuelve a habilitar las interrupciones (recordemos que al principio de todo, ejecutamos la instrucción cli por lo que las interrupciones habían quedado deshabilitadas).

3.4. Ejercicio 4

Para inicializar el directorio y las tablas de páginas para el kernel, en el archivo *kernel.asm* llamamos a la función *mmu_initKernelDir* que se encuentra en *mmu.c*.

Además. en este archivo *mmu.c* se encuentran las siguientes funciones (que serán utilizadas por *mmu_initKernelDir*):

mmu_nextFreeTaskPage() que actualiza la variable *proxima_pagina_libre_task* (le suma PAGE SIZE, que vale 4K) y retorna la dirección de una página libre destinada a una tarea.

mmu_nextFreeKernelPage() que es análoga a *mmu_nextFreeTaskPage* pero esta vez actualiza *textitproxima_pagina_libre_kernel* y retorna una página destinada al kernel.

Además, en *mmu.h* definimos ciertas macros que serán explicadas al ser usadas y las estructuras **pde** y **pte**, que guardan los campos de una *page directory entry* y una *page table entry* correspondientemente.

Una vez explicadas estas funciones auxiliares, retomamos *mmu_initKernelDir*. Esta función se basa en la variable *dir*, que es un arreglo de **pde** y se encuentra en la posición *KERNEL_PAGE_DIR* (macro que vale 0x2B000 según la consigna). En cada entrada de *dir* (son 1024, la macro *PAGE_DIRECTORY_SIZE* está en *mmu.h*) seteamos el bit *present* en 0. Esto es porque inicialmente todas las **pde** salvo la primera no se usan. En cambio para *dir[0]* seteamos los campos:

present = 1

readwrite = 1

usersupervisor = 0

pwt = 0

pcd = 0

accesed = 0

ignored = 0

ps = 0

g = 0

disponible = 0

direccion_tabla_de_descriptores_de_paginas = KERNEL_PAGE_TABLE>> 12. Esta macro vale 0x2C000 porque decidimos que la tabla de páginas esté a continuación del directorio de páginas, que ocupa 4K.

Como necesitamos 4MB para el Kernel, basta un **pde** (es decir, una sola tabla de páginas) ya que este abarca 1024 **pte's**, y cada una referencia a una página que ocupa 4KB. Es decir, con una sola **pde** tendremos 4kB * 1K, es decir 4MB.

Por lo explicado recién basta con completar los campos de las **pte** de la única tabla de páginas del kernel. Para esto, haremos un ciclo iterando con una variable i, que irá desde 0 hasta PAGE_TABLE_SIZE (macro que vale 1024) exclusive. Los campos son:

present = 1

readwrite = 1

usersupervisor = 0

pwt = 0

pcd = 0

accesed = 0

dirtybit = 0

pat = 0

g = 0

disponible = 0

direccion_pagina = i. Para más información, ver los generosos y didácticos comentarios que se encuentran en esta función.

Volviendo a *kernel.asm* y luego de llamar a *mmu_initKernelDir*, nos encargamos de cargar el directorio de páginas. Esto es cargar *KERNEL_PAGE_DIR* en el registro **cr3**. Luego para habilitar paginación (**cr0.P** = 1) modificamos el bit más significativo de los 32 bits de **cr0**.

Además, a esta altura del trabajo, imprimíamos por pantalla nuestros números de libreta (no está claro el motivo, probablemente sea para testear la paginación a esta altura del trabajo práctico). Esto lo hacíamos con la macro *"print_text_pm LU, LU_size, 0x07, 0, 0"*, pero luego lo comentamos para no dejar escrita una zona correspondiente al tablero.

3.5. Ejercicio 5

Para inicializar las estructuras necesarias para administrar memoria en el área libre del kernel, efectuamos una llamada a la función *mmu_init*. Esta inicializa las variables *proxima_pagina_libre_kernel* y *proxima_pagina_libre_task*. Para esto, utiliza las macros definidas en *mmu.h*: *INICIO_PAGINAS_LIBRES_KERNEL* e *INICIO_PAGINAS_LIBRES_TASK*, que valen 0x100000 y 0x400000.

Luego, escribimos las funciones *mmu_mapPage* y *mmu_unmapPage* que se encargan de mappear y desmappear páginas a gusto y piacere.

mmu_mapPage recibe la dirección virtual, su **cr3** correspondiente, la dirección física a la cual se mappeará y los atributos deseados. Hemos decidido que solo importan los bits **user/supervisor** y **read/write**, por lo que del parámetro *atrrs* solo usamos los dos bits menos significativos (el bit 0 funciona como **user/supervisor** y el bit 1 funciona como **read/write**).

A partir del **cr3** (que contiene la dirección del directorio en sus 12 bits más significativos), la dirección virtual recibida y la dirección física recibida, determinamos los offsets utilizados para la paginación correspondiente. La virtual recibida tiene en los bits 22 a 31 el offset dentro del directorio, por lo que para obtenerlo shifteamos la virtual 22 bits a la derecha. Además en los bits 12 a 21 el offset dentro de la tabla de páginas así que nos quedamos con este valor al shiftear la virtual 10 bits a la izquierda y 22 a la derecha. El offset dentro de la página no es de interés.

Si la página no estaba mappeada aún, en esta parte del código nos encargamos de hacerlo pidiendo una página del área libre del kernel e inicializándola con todos sus **pte's** no presentes. Ahora actualizamos los campos del **pde** correspondiente a la tabla recién creada. Notar que el bit *present* siempre valdrá 1 mientras que todos los demás que no sean *readwrite* ni *usersupervisor* valdrán 0.

Ahora independientemente de si fue necesario mappear la página o no, llegamos a la **pte** correspondiente y ponemos el bit *present* en 1, los campos *readwrite* y *usersupervisor* según lo recibido por parámetro, el campo *dirrecion_pagina* como la dirección física recibida pero shifteada 12 a la derecha (esto es para que sea múltiplo de 4k) y el resto de los atributos en 0.

Por suerte, la función *mmu_unmapPage* es mucho más fácil de explicar. Recibe la dirección virtual a desmappear y el **cr3** para conocer el directorio, desglosa la virtual para obtener los offsets y una vez que tenemos acceso a la *page table entry* correspondiente, ponemos el bit *present* $= 0$. Esta función devuelve un valor de tipo *uint32_t* para el cual no encontramos utilidad. Es por esto que decidimos que devuelva 0.

Finalmente, escribimos la función *mmu_initTaskDir*, que decidimos que reciba un parámetro llamado tipoDeTarea. Si tipoDeTarea es 0, se refiere a jugador A tipo 1. Si tipoDeTarea es 1, es jugador A, etc. Si tipoDeTarea es 3, es jugador B tipo 1, etc. Si tipoDeTarea es 5, es jugador B tipo 3.

Aquí pedimos memoria al kernel para el **cr3** de la tarea, mappeamos al principio de la tarea toda la memoria correspondiente al kernel mediante identity mapping y nos guardamos el **cr3** actual. Ahora reservamos memoria del área libre de tareas, llamemos a esta página *nuevaPaginaDelAreaDeTareaCodigo*, como en el algoritmo. Queremos escribir esta nueva página y además asignarle una página para su stack, pero no podemos escribirla si no está mappeada en el **cr3** actual, ese que guardamos. Es por esto que, además de mappear la página de tarea para código (y su página de stack) usando el **cr3** de la tarea en cuestion, también mappeamos estas dos páginas usando el **cr3** actual. De otro modo, estas páginas no podrían ser modificadas.

Además usamos la función *obtenerPosicionDeMemoriaDeCodigo*, que devuelve la dirección de memoria dentro del área libre del kernel en la cual se encuentra el código de la tarea según su tipo. Así, copiamos toda esta página y luego desmappeamos las páginas utilizadas en el paso intermedio. La función retorna

el **cr3** de la tarea que acaba de ser creada.

También realizamos el test solicitado en el punto 5.d, que quedó comentado en la línea 224 de *kernel.asm*.

3.6. Ejercicio 6

Al tener tareas independientes, cada una necesita tener un segmento definido y que este sea de tipo TSS. Es por esto que, volviendo a *gdt.c*, le agregamos a la tabla de descriptores ciertas entradas:

GDT_TSS_TAREA_INICIAL, GDT_TSS_IDLE,

GDT_TSS_*A1*, ..., GDT_TSS_*A3*,

GDT_TSS_HANDLER_*A1*, ..., GDT_TSS_HANDLER_*A3*,

GDT_TSS_*B1*, ..., GDT_TSS_*B3*,

GDT_TSS_HANDLER_*B1*, ..., GDT_TSS_HANDLER_*B3*.

Notar que en todas estas los campos son los mismos, por lo que los explicamos una sola vez:

- Limite = 0x67 siempre ya que esta cantidad de bytes permite guardar todos los registros pertinentes a una TSS entry.
- Type = 0x09, es decir TSS not busy (b = 0).
- Base = 0x00. Luego será modificada, cuando tengamos las tss declaradas.
- DPL = 0 siempre ya que una tarea no puede llamar a otra tarea.
- Present = 1.
- AVL = 0.
- G = 0.
- el bit que correspondería al atributo System es siempre 0, declarando así que este segmento no es de código ni de datos.

La tarea *Idle* no modifica las pelotas ni el tablero sino que dibuja un simpático reloj a modo de stand by. Por esto, en la función *tss_init*, haremos que todos los campos de su tss sean 0 salvo los siguientes:

- **eip:** 0x0001C000, por consigna la tarea se encuentra en esta dirección, y además no utilizará stack. Entonces no le dejamos espacio al stack.
- **eflags:** 0x202 para permitir interrupciones.
- **esp** y **ebp:** 0x2B000 ya que su pila comparte la dirección de la pila del kernel.
- **cs:** GDT_CODIGO_KERNEL $<<$ 3 ya que es una tarea de nivel 0.
- **el resto de sus selectores de segmento:** GDT_DATOS_KERNEL $<<$ 3 por la misma razón.

En esta misma función *tss_init* aprovechamos para incializar el arreglo *kernelLevelTasksStacks*, llamando para cada una de sus 6 entradas a *mmu_nextFreeKernelPage* de modo tal que cada una de las tareas tenga una pila de nivel cero en el área libre del kernel.

Además, para cada una de las entradas de la GDT correspondientes a una TSS, llenamos el campo Base. Esto lo realizamos en esta función porque recién en este archivo *tss.c* definimos las distintas *tsss* como variables globales y aquí podemos tomar sus direcciones de memoria para llenar los campos Base correspondientes.

Luego programamos la función *initUserTask*, que se encarga de completar una tss libre con los datos correspondientes a una tarea. Para esto, la función recibe *slotDeTarea* (vale entre 0 y 2 para los slots 1 a 3 del jugador A, y vale entre 3 y 5 para los slots 1 a 3 del jugador B), recibe *esHandler* (utilizada como

un booleano, vale 1 si la tarea a iniciar es un handler de una de las tareas) y también recibe *punteroARutinaHandler*, que lo usamos solo cuando *esHandler* es efectivamente 1.

Efectuando un switch con el slot recibido por parámetro, y viendo en cada caso si la tarea en cuestión es handler, llamamos correspondientemente a la función que llena los campos. Para esto programamos un conjunto de funciones del estilo *llenarTSSA1* y *llenarTSSA1_Handler* para los tres slots de los dos jugadores.

En este conjunto de funciones, definimos todas las macros que son visibles al comienzo del archivo *tss.c*. Como las tareas y los handlers comparten la gran mayoría de los campos, los nombres de muchas macros comienzan con *" tasksAndHandlers_"* seguido del nombre del campo en cuestión. Para el resto de los campos o bien comienzan con *"tasks_"* o bien con *"handlers_"* seguidos del campo correspondiente.

Campos relevantes (no nulos) de las tss's de las tareas (no idle ni inicial) y sus handlers:

- **tasksAndHandlers_ss0:** (GDT_DATOS_KERNEL << 3) el stack de nivel 0 de las tareas, tendrá el segmento de kernel.
- **tasks_eip:** 0x08000000 porque todas las tareas estan mapeadas desde 0x08000000 mediante la funcion mmu_initTaskDir
- **tasksAndHandlers_eflags:** 0x202, para que las interrupciones estén activadas.
- **tasks_esp y tasks_ebp:** definimos los ebp por si acaso en lo mismo que el esp (0x08000000 + 7*1024)
- **handlers_esp y handlers_ebp:** definimos los ebp por si acaso en lo mismo que el esp (0x08000000 + 8*1024)
- **tasksAndHandlers_ es, ss, ds, fs, gs:** (GDT_ DATOS_ USUARIO << 3) + 3, nótese que RPL lo ponemos en 11 porque vamos a estar en nivel 3.
- **tasksAndHandlers_ cs:** (GDT_ CODIGO_ USUARIO << 3) + 3 , nótese que RPL lo ponemos en 11 porque vamos a estar en nivel 3.
- **tasks_ esp0:** kernelLevelTasksStacks[slotDeTarea] + 2 * 1024 - 1 , en kernelLevelTasksStacks teníamos las paginas de pila de nivel 0 por slot. Las tareas y los handlers comparten una sola página de stack. La tarea usa los primeros 2 kB (mirar figura 5 del enunciado del tp)
- **handlers_ esp0:** kernelLevelTasksStacks[slotDeTarea] + 4 * 1024 - 1 (mismo motivo)
- **tasks_ cr3:** userLevelTasksCodeAndStacks[slotDeTarea] aquí guardábamos los cr3 de cada tarea que creamos utilizando mmu_ initTaskDir en la función iniciarTarea (en game.c)
- **handlers_ cr3:** para el handler de A1, será tss_ A1.cr3 , porque los handlers comparten cr3 con sus tareas.
- **handlers_ eip:** recibimos esto como parámetro en la función llenarTSSA1_ Handler, que lo llamará la syscall que inicia la tarea con el puntero a la función.

Volviendo a *kernel.asm* y luego de llamar a *tss_init*, cargamos manualmente la tarea inicial *GDT_TSS_TAREA_INICIAL* mediante poner en **ax** el valor *GDT_TSS_TAREA_INICIAL* << 3 (ya que **RPL** = 00, **TI** = 0) y luego cargar **ax** en el *task register* mediante la instrucción **ltr ax**.

Más adelante saltaremos de la tarea inicial a *Idle*, pero para esto primero inicializaremos todas las estructuras correspondientes al juego en sí. Solo después de esto, podremos realizar el salto mediante la instrucción *jmp (20<<3):0*.

3.7. Ejercicio 7

Para la inicialización del scheduler, desde *kernel.asm* realizamos una llamada a *sched_init*, que vive en *sched.c*. Aquí definimos que el valor **quantum** comienza en -1 ya que antes de ser utilizado se incrementa siempre, por lo que la primera vez valdrá 0. Además inicializamos los arreglos *handlers_activos*, *pelotas_vivas* y *ejecutando_handler*, todos con cada una de sus entradas en cero ya que inicialmente no hay nada vivo ni corriendo, esto es previo al Big Bang.

La función *sched_nextTask()* devuelve un entero de 16 bits que es el selector de segmento de la tarea que debe ejecutarse a continuación. Esto es práctico para realizar el *jmp far* con lo que devuelva esta función.

Para que *sched_nextTask()* pueda cumplir su cometido, hemos hecho una serie de funciones auxiliares:

- **next_quantum** Que incrementa y devuelve el valor de la variable **quantum** tomándole módulo 7
- **matarTarea** Que desaloja una tarea y actualiza las estructuras del juego (pone los valores necesarios en 0 como estaban inicialmente).
- **GDTEntryBySlot** Que devuelve el índice de la GDT de la tarea correspondiente al slot recibido por parámetro. Consiste en un switch con 6 casos.
- **GDTHandlerEntryBySlot** Que análogamente devuelve el índice de la GDT del handler correspondiente al slot recibido por parámetro.

Volviendo a la explicación de *sched_nextTask*, esta pide el próximo **quantum** con *next_quantum* y en base a lo recibido opera. Si este valor es 6 entonces en el ciclo actual corresponde dibujar la pantalla, por lo que *sched_nextTask* comunicará esto devolviendo 0. Luego en la RAI del clock se manejará pertinentemente este caso.
En cambio si **quantum** no es 6, primero revisamos si el handler de la tarea en cuestión aún se estaba ejecutando y no terminó en su clock correspondiente. En tal caso, llamamos a *matarTarea(quantum)* para desalojarla inmediatamente.

Si el handler estaba activo (si la tarea había llamado a *setHandler*) y el juego no se encuentra pausado, entonces reiniciamos dicho handler, es decir, actualizamos *ejecutando_handler[quantum]*, que ahora valdrá 1, y también volvemos a inicializar el handler en cuestión mediante *initUserTask(quantum, 1, handlers_activos[quantum])*. Resta devolver la entrada de la **gdt** asociada a dicho handler, para lo cual llamamos a *GDTEntryBySlot(quantum)* y a lo que devuelva lo shifteamos 3 bits a la derecha.

En cambio si no está activo el handler pero la pelota está viva, significa que esta pelota aún no ha llamado a *setHandler*, por lo que el scheduler hará que la próxima tarea a ejecutar sea la de esta pelota y punto. Esto lo realizamos mediante devolver GDTEntryBySlot(quantum) $<<$ 3.
Si no ocurre ninguna de las condiciones anteriores, entonces el juego está en pausa o no hay handler activo ni pelota viva. En este caso, le toca ejecutar a *Idle* ya que no hay más para hacer. Devolvemos GDT_TSS_IDLE $<<$ 3.

Como habíamos prometido, modificamos la rutina de la *interrupción 0x47*, que hasta este punto del trabajo práctico solamente modificaba el valor de **eax** arbitrariamente. Ahora implementará los distintos servicios del sistema según la sección 3.1.

Comenzando por preservar todos los registros mediante la instrucción pushad, la función *_isr47* definida en *isr.asm* consiste en leer el código del servicio y actuar según el caso. Este código se encuentra en el registro **eax** y para cada uno de sus posibles valores se realiza la llamada a la función auxiliar que corresponda.

Para esto hemos hecho las siguientes funciones en ASM:

- **setHandler**: consiste en llamar a *setHandlerValue*, pasándole por parámetro mediante la pila lo que originalmente se encontraba en **ebx**, es decir, el puntero al handler. La función *setHandlerValue* se encuentra en *sched.c* y lo único que hace es guardar en *handlers_activos[quantum]* el puntero recibido.

- **talk:** llama a *write_message*, que se encuentra en *game.c* y recibe por parámetro el string a escribir. Según el **quantum**, decide si el mensaje es del jugador A o B y luego lo inserta en el arreglo *mensajesJugadorA* o *mensajesJugadorB* correspondientemente. Esta inserción funciona como un ”push” ya que el nuevo mensaje aparece en la posición 0 y el mensaje que se encontraba en la posición 2 deja de tener lugar. Para la copia hicimos la función *copiarString*, que simplemente copia char a char e insereta el caracter nulo al final.

- **where:** llama a *dameCoordenadas*, función en *game.c* que devuelve un puntero a la estructura de tipo *coordenadaPelota* que se encontraba en *coordsPelotasPorSlot[quantum]*. Luego, nuevamente en assembler, tenemos dicho puntero en **eax**. Resta mover a **ebx** y a **ecx** los valores de las coordenadas **X** e **Y** respectivamente (los offsets son 0 y 4 ya que son los dos primeros campos dentro de la estructura y ambos ocupan 4 bytes).

- **informAction:** llama a *actualizarMovimientoPendiente*, que está en *game.c*. Esta última solamente pone en *movimientosPendientesPorSlot[quantum]* la acción recibida por parámetro. Luego, en el séptimo **quantum**, se dibuja y se actualiza el tablero en función de los movimientos pendientes que cada pelota tiene guardada aquí en este arreglo. Volviendo a assembler y luego de llamar a *actualizarMovimientoPendiente*, realizamos una llamada a *saltarDeHandlerATarea*. Esta última función está en *sched.c* y se encarga de informar que ya no se está ejecutando el handler de la tarea correspondiente (mediante ejecutando_handler[quantum] = 0) y saltar a la tarea en cuestión mediante saltarATarea(GDTEntryBySlot(quantum) << 3). Por su parte, *saltarATarea* es una función auxiliar en *kernel.asm* que recibe por pila el selector de segmento al cual hay que saltar, así que realiza un *jmp far* a dicho selector siempre y cuando no sea el mismo almacenado en el *task register*.

Como prometimos, modificamos la rutina de atención del clock de forma tal que se encargue de intercambiar tareas cada vez que transcurre un ciclo de clock. La rutina en cuestión es la de la función *_isr32*, que llama a *sched_nextTask*. Como se explicó en la subsección del ejercicio 6, esta función auxiliar devuelve el selector de segmento de la próxima tarea a ser ejecutada así que en el registro **ax** tendremos dicho valor.

Para no intentar saltar a la misma tarea que se está ejecutando actualmente, revisamos si *ax* es igual al valor actual guardado en el *task register*. En tal caso, saltamos a la etiqueta *.fin*, donde restauramos los registros preservados y ejecutamos *iret*.

Recordemos que si el selector de segmento devuelvo vale 0, entonces estamos en el séptimo **quantum** por lo que toca dibujar la pantalla. Para esto programamos la función *dibujarPantalla*, que se encuentra en *game.c*. Si *ax* no es 0 entonces al hacer **”mov [selector], ax”** y luego **”jmp far [offset]”** estamos saltando correctamente a la tarea que había indicado *sched_nextTask*.

Retomando *dibujarPantalla*, esta función imprime (siempre que *enPausa* valga 0) todos los datos correspondientes al Pong. Para esto primero limpia el tablero (imprime el mismo rectángulo de 70x40 que sirve de tablero) y luego ejecuta un ciclo de 6 iteraciones donde cada iteración se corresponde con una de las 6 pelotas.

Dentro del ciclo y para cada i, se actualiza el valor de *coordsPelotasPorSlot[i]* en base al movimiento pendiente que tenga dicha pelota. Para esto se invierte *movimientosPendientesPorSlot[i]* según el valor *direcciónY* y luego a este resultado se le aplica *moverEnVertical* y *moverEnHorizontal*, que devuelven un valor de tipo *coordenadaPelota* luego de mover correspondientemente y teniendo en cuenta los bordes del tablero. Luego de todo esto, se imprime un asterisco para representar a la pelota, que tendrá el color del

jugador que la lanzó.

Luego de este ciclo, limpiamos las columnas 0 y 79 en las que pueden moverse los jugadores A y B respectivamente y volvemos a imprimir a los jugadores ya que las variables *alturaJugadorA* y *alturaJugadorB* podrían haber sido modificadas en caso de haberse pulsado las teclas de movimiento de los jugadores, por lo que hay que actualizar las posiciones de los jugadores.

Después de dibujar a los jugadores, limpiamos la zona donde van los mensajes e imprimimos en su totalidad a los arreglos *mensajesJugadorA* y *mensajesJugadorB*.

A continuación imprimimos los puntajes de cada jugador. Notar que la zona donde van los puntajes ya ha sido limpiada para poder imprimir los mensajes, por lo que basta con imprimir los valores de *puntajeA* y *puntajeB*.

Adicionalmente, por cada uno de los 3 slots de ambos jugadores, imprimimos el caracter 'O' si la pelota está viva (es decir, si *pelotas_vivas[i]* != 0) o el caracter 'X' en caso contrario.

Finalmente, imprimimos el mensaje **"Modo debug"** si este modo está activado, es decir, si la variable *modoDebug* no es cero.

Volviendo a *_isr32* y en caso de haber llamado a *dibujarPantalla*, la próxima acción es saltar a la tarea *Idle* siempre y cuando no sea esta tarea la que ya se estaba ejecutando antes de la interrupción. En tal caso, basta con ejecutar la instrucción jmp (20<<3):0.

Habíamos prometido modificar las rutinas de excepciones para que realmente se hagan cargo de desalojar a las tareas en caso de ser necesario. Ahora, cada una de las 32 rutinas de excepciones consisten en llamar a *manejar_excepcion*, realinear la pila y luego haltear mediante *jmp $*. El problema estético es que *manejar_excepcion* recibe 24 parámetros, por lo que debemos pushear 24 valores. Cada uno de estos es un valor que debe ser impreso en el modo debug.

Una vez en *manejar_excepcion*, que se encuentra en *game.c*, revisamos si la excpeción se produjo durante la ejecución de alguna de las tareas. Esto lo podemos saber viendo si **quantum** es menor que 6. En caso positivo, debemos desalojar esta tarea. Pero si estaba el *modo debug activo*, imprimimos por pantalla los 24 valores recibidos lo más user-friendly posible. Todo el código entre las líneas 250 y 386 está destinado a estas diversas impresiones, y consideramos que es lo suficientemente declarativo gracias a los nombres de las variables y los comentarios.

Luego de imprimir la información pertinente al modo debug (o no), resta desalojar la tarea y saltar a la tarea *Idle*. Para el desalojo llamamos a *matarTarea(quantum)*, mientras que para el salto llamamos a *saltarATarea*(GDT_TSS_IDLE << 3). La explicación de ambas funciones se encuentran al principio de esta misma subsección.

Adicionalmente, modificamos la rutina de atención *_isr33*. Hasta aquí, imprimía un inocente número entre 0 y 9 si dicha tecla fue presionada mediante el llamado a la función *print_number*. Ahora en lugar de esto, llama a la función *atender_teclado*, que se encuentra en *game.c*. Aquí dentro tenemos un switch que depende del valor recibido por parámetro: el código de la tecla. Para que sea más comprensible el código, definimos al principio de *game.c* macros para cada una de estas teclas de interés, como por ejemplo CODE_w que vale *0x11*.

Si la tecla presionada es de movimiento, se llama a *moverJugador*. Esta función recibe un entero usado como variable booleana ya que si es 1 significa que el jugador a mover es el jugador A y 0 si es el jugador B. Además recibe el movimiento a ejecutar, modificando la variable *alturaJugadorA* o *alturaJugadorB* correspondientemente y teniendo en cuenta los bordes de la pantalla.

En cambio, si la tecla presionada corresponde al lanzamiento de una pelota, llamamos a la función *crearPelota*, que también recibe un booleano (vale 1 si corresponde al jugador A o vale 0 si corresponde al jugador B) y un número entre 0 y 2. El número 0 indica tipo 1, el número 1 indica tipo 2 y el número 2 indica tipo 3.

La función *crearPelota*, también en *game.c*, llama a *dameSlotLibre* para saber qué slot le corresponde a la pelota nueva, llamemos *slotLibre* a este valor. Luego verifica si el jugador en cuestión tiene pelotas disponibles (verifica si pelotasDisponiblesA > 0 o verifica si pelotasDisponiblesB > 0 según corresponda) y en tal caso decrementa pelotasDisponiblesA (o pelotasDisponiblesB según corresponda) y luego inicializa los campos de coordsPelotasPorSlot[*slotLibre*] según el jugador que creó la pelota y su altura.

Luego, se llama a iniciarTarea. Esta función también está en *game.c* y modifica el valor de *pelotas_vivas[slotLibre]* poniéndole 1. También reserva una página para el **cr3** de esta nueva tarea llamando a *mmu_initTaskDir* y guarda este **cr3** en *userLevelTasksCodeAndStacks[slotLibre]*. Finalmente, llama a *initUserTask*, función explicada en la subsección del ejercicio 6.

Una vez llegado a este punto, tenemos todo lo necesario para que el juego pueda correr sin problemas.

4. Conclusión

A lo largo de este trabajo práctico, hemos atravesado todos los conceptos teóricos dados en la segunda parte de la materia. Sin duda alguna, este trabajo nos permitió asentar todos los conocimientos que poseíamos y permitió dar ese salto entre la teoría y la práctica.

Dada la complejidad de programar un sistema operativo en la vida real (basta con pensar en alguna distribución de GNU/Linux), consideramos que esta experiencia fue completamente fructífera ya que tuvimos la posibilidad de lidiar con problemas de la vida real tales como *General Protection Fault*.

Además hemos enfrentado el manejo de memoria como nunca antes en la carrera. La no existencia funciones como malloc o calloc nos forzó a programar cuidadosamente cada variable y estructura.

Por ejemplo, hemos podido observar la diferencia entre declarar **"char mensajesJugadorA[3][21]"** y declarar **"char *mensajesJugadorA[3]"** dado que la primera opción reserva 3 posiciones en las cuales se guardan arreglos de 21 caracteres, y la segunda opción reserva 3 posiciones en las cuales se guardan un puntero a char. Esta última opción nos provocaba que al escribir por ejemplo en mensajesJugadorA[3] y sobrepasar el tamaño reservado para un puntero, terminábamos pisando la memoria de otras variables que no tenían nada que ver con este arreglo. Y por supuesto, al poseer nosotros control absoluto, el simulador no lo considera un error ya que es toda la memoria es nuestra, nos pertenece dado nuestro rol de Kernel. Detalles de este estilo sin duda han enriquecido nuestra comprensión de la memoria y del bajo nivel en sí mismo.