DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Organización del Computador II

Segundo Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Rodrigo Laconte	193/18	rola1475@gmail.com
Julia Rabinowicz	48/18	julirabinowicz@gmail.com
Amalia Sorondo	281/18	sorondo.amalia@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

En este trabajo diseñamos y programamos un Sistema Operativo que permite correr 12 tareas, implementando un juego conocido como Pong. En la siguiente sección describiremos cómo decidimos implementar las estructuras necesarias respetando las especificaciones de las consignas.

2. Desarrollo

2.1. Pasaje a Modo Protegido

Cuando se inicia el equipo el CPU ejecuta en Modo Real, por lo que para pasar a Modo Protegido los pasos a seguir fueron: habilitar A20, llamando a la función `A20_enable` provista por la cátedra, cargar la GDT (estructura que detallaremos en la sección siguiente) en el registro GDTR, setear el bit CR0.PE y finalmente realizar el jmp far a Modo Protegido.

2.2. Global Descriptor Table

En un principio llenamos las primeras 5 entradas de la GDT siguientes a la 13, ya que la consigna explicitaba que empezáramos desde la 14. La primer entrada, es decir la de índice 0, ya estaba completa con el segmento nulo, por lo que nuestro primer paso fue completar 4 segmentos (dos de código y dos de datos, correspondientes a nivel 0 y 3). Para establecer las bases y límites utilizamos segmentación flat, de manera que todos comienzan en la dirección 0 y finalizan en la 0xA2FFFFF. Este límite se debe a que debíamos direccionar los primeros 163MB de memoria. Luego, en el índice 18, agregamos el segmento descriptor del área de pantalla.

Los campos de los segmentos mencionados quedaron de la siguiente manera:

	Indice	Base	Límite	DPL	Tipo	S	G
Código nivel 0	14	0x0	0xA2FF	0	0xA	1	1
Datos nivel 0	15	0x0	0xA2FF	0	0x2	1	1
Código nivel 3	16	0x0	0xA2FF	3	0xA	1	1
Datos nivel 3	17	0x0	0xA2FF	3	0x2	1	1
Pantalla	18	0x8000	80*50*2-1	0	0x2	1	0
Tarea Idle	19	*	0x67	0	0x9	0	0
Tarea inicial	20	*	0x67	0	0x9	0	0
Tarea jug 1 tipo 1	21	*	0x67	0	0x9	0	0
Handler tarea jug 1 tipo 1	22	*	0x67	0	0x9	0	0
...							
Tarea jug 2 tipo 3	30	*	0x67	0	0x9	0	0
Handler tarea jug 2 tipo 3	31	*	0x67	0	0x9	0	0

Cuadro 1: Selectores de segmento contenidos en la GDT

Los descriptores de TSS se crean en tiempo de ejecución gracias a la función `completarGDT`, contenida en el archivo tss.c. Por este motivo, las bases marcadas con * están determinadas por la dirección base de las TSS al definirlas al inicio de tss.c. Hablaremos más en detalle sobre el tema en la sección 2.5.

2.3. Modo Protegido

Una vez finalizado el pasaje a Modo Protegido pusimos el selector de segmento de código de nivel 0 en los registros de segmento DS, GS, FS y SS. Además establecimos como base de la pila del kernel la dirección 0x2B000, e inicializamos la pantalla poniendo en el registro ES el valor 0x90.

2.4. Memory Management Unit

La MMU provee distintas funciones que podrá utilizar el Kernel para mapear y desmapear páginas de cualquier directorio de páginas dentro de la memoria. Lo primero que realiza el Kernel es inicializar el manejador de memoria llamando a `mmu_init`. Luego inicializa el directorio de páginas llamando a `mmu_initKernelDir`. Después carga el directorio de páginas con el valor que devuelve esta úlitma función y finalmente habilita paginación modificando el bit CR0.PG. Ahora vamos a analizar cada una de las funciones que provee esta unidad:

2.4.1. mmu_init

Esta función se encarga de ubicar la primera posición del área libre del Kernel, cuyo valor es 0x100000, y asignarla a la variable llamada ”proxima_pagina_libre_kernel”. Al pedir una nueva página se actualizará esta variable de forma que apunte siempre a la próxima página libre del Kernel. De la misma forma, se le asigna a la variable ”proxima_pagina_libre_task” el valor 0x400000.

2.4.2. mmu_nextFreeKernelPage

Esta función devuelve la dirección apuntada por la variable ”proxima_pagina_libre_kernel”. Al pedir una nueva página se debe actualizar la variable de forma que apunte siempre a la próxima página libre del Kernel, por lo tanto se la incrementa 4KB (una página).

2.4.3. mmu_nextFreeTaskPage

Funciona de forma análoga a la función explicada anteriormente pero devolviendo la dirección apuntada por ”proxima_pagina_libre_task” e incrementando esta variable en 4KB.

2.4.4. mmu_mapPage

La función toma como parámetros una dirección virtual, la dirección de un directorio de páginas (cr3), una dirección física, el atributo user/system y el atributo read/write. Esta función se encarga entonces de mapear la dirección física al directorio de páginas con la dirección virtual pasada como parámetro respetando los atributos especificados. Para realizar lo anterior primero se verifica si el índice correspondiente, determinado por la dirección virtual pasada como parámetro, en el directorio de páginas tiene el bit de presente en 1, si no es el caso se solicita una nueva página libre del Kernel y se define la entrada de tabla de páginas que se requiere. Luego se accede a la tabla de páginas en el índice que corresponde y se la completa de acuerdo con la dirección física y los atributos pasados como parámetro.

2.4.5. mmu_unmapPage

A su vez, esta función será la encargada de desmapear una página física a partir de una dirección virtual y un CR3. Para lograr esto descompone la dirección virtual recibida y a partir del índice en el directorio de páginas accede a la tabla de páginas y pone el bit de presente en 0 de la entrada correspondiente.

2.4.6. mmu_initKernelDir

La función se ocupa de inicializar el directorio de tablas de páginas del Kernel. Este directorio va a tener mapeada toda la memoria correspondiente al Kernel y a su área libre mediante identity mapping. El directorio se encontrará en la dirección 0x2B000, este valor se cargará en su cr3. El kernel y su área libre ocupan 4MB, luego solo necesitamos una entrada de la PD y mapear con identity mapping las 1024 páginas contenidas en la tabla de páginas apuntada por la entrada mencionada.

2.4.7. mapearTareas

La función mapearTareas realiza un ciclo donde para cada tarea primero mapea con identity mapping el área del Kernel, luego pide dos páginas que serán utilizadas para copiar el código de la tarea cuando ésta se cree. Guardamos la dirección de la primera página (la segunda página es contigua en memoria) en el arreglo PAG_CODIGO. Este arreglo sirve para utilizar para cada tarea siempre las mismas páginas para guardar el código, aunque éste se irá actualizando, y así no pedir nuevas páginas al crear tareas. Finalmente se mapean las dos páginas creadas al directorio de tablas de páginas de la tarea, en las direcciones virtuales 0x08000000 y 0x08000000 + 0x1000.

2.4.8. copiarCodigo

Esta función recibe un entero que representa el jugador cuyo valor puede ser igual a 1 o a 2, un tipo de tarea y un índice que representa en cuál tarea, ya inicializada gracias a la función anterior, se va a copiar el código de la tarea. La función va a ser llamada al crearse una tarea y va a servir para copiar el código de la tarea que corresponde en las páginas ya mapeadas y reservadas para este propósito. Entonces, lo primero que se hace es mapear las páginas de memoria física reservadas a la memoria del cr3 actual, luego se copian las dos páginas en la memoria física de las páginas reservadas y finalmente se desmapea lo que se acaba de mapear.

2.5. Tareas

Lo siguiente que realiza el Kernel es llamar a la función `tss_init` para inicializar las TSS. En el archivo tss.c se cuenta con todas las funciones necesarias para administrar las estructuras de las tareas. Cada tarea posee un Segmento de Estado (TSS), en donde se almacena el contexto de la tarea al ser interrumpida por el reloj para luego poder reanudarla en el mismo estado. A su vez, cada pelota tiene su propio handler. Éste es una función dentro del código de la tarea que actúa como intermediario entre la misma y el sistema. También necesitamos una tarea inicial para saltar a la primera tarea y una tarea Idle que se ejecutará cuando alguna tarea no se esté ejecutando en el momento en el que cae la interrupción de reloj y al finalizar cada ciclo del scheduler. Luego, como ya vimos en la tabla 1, en la GDT tendremos 14 descriptores de TSS. Para completar estas entradas necesitamos información sobre las direcciones físicas en donde van a encontrarse las tareas. Por este motivo optamos por definirlas en tiempo de ejecución. Las TSS se definen entonces en el inicio del archivo tss.c, y las tareas se inicializan con basura que luego pisaremos. Definimos distintos arreglos para facilitar la creación y el acceso a la información de las tareas. Para poder modificar fácilmente las estructuras creamos el arreglo de punteros a TSS, TSS_ARRRAYS, que posee en la primera posición un puntero a la tarea Idle, luego las 6 tareas en orden seguidas por los 6 handlers correspondientes. A su vez, definimos el arreglo handlers que si éste ya fue seteado por la tarea contiene el selector de TSS del handler, y si no contiene un 0. Además, creamos un arreglo ”pila0_tareas” que como su nombre lo indica, va a contener punteros a las pilas de nivel 0 de cada tarea. Por último, guardamos en la i-ésima posición del arreglo slots_disponibles un 1 si la tarea relacionada con dicho índice se encuentra en el scheduler y un 0 en caso contrario. En todos los arreglos de tamaño 6, las primeras 3 posiciones corresponden a las tareas del jugador 1 y las 3 últimas a las tareas del jugador 2. A continuación detallamos las funciones contenidas en tss.c que sirven al manejo de las tareas.

2.5.1. tss_init

Primero se pide para cada tarea una página para guardar su pila de nivel 0 y se guarda la dirección física de ésta en el arreglo ”pila0_tareas”. A continuación se completa el arreglo TSS_ARRRAYS con las direcciones que obtuvimos al definir las TSS y se completan con su valor inicial otras estructuras que mencionaremos más tarde al analizar el comportamiento del Scheduler. La función se encarga de completar los campos de la TSS de la tarea Idle y finalmente llama a `completarGDT` cuya única función es inicializar los descriptores de TSS de todas las tareas. Como podemos ver en la tabla 1, elegimos setear el límite en 0x67 ya que este es el mínimo límite para una TSS, el tipo igual a 9 y S en 0 ya que se trata de descriptores de TSS, la granularidad de todos éstos es igual a 0.

2.5.2. tss_init_task

Esta función es la encargada de llenar la información de una TSS cuando el usuario decide crear una nueva tarea. Recibe como parámetros tres enteros; el primero representa el jugador, el segundo el tipo de la tarea solicitada y el último es el índice de la misma. Esta función es llamada cuando se sabe que el jugador está en condiciones de solicitar una nueva tarea, por lo que no se realiza ninguna verificación. En un primer lugar, la función mapea la pila de nivel 0, luego completa los campos de las TSS de la tarea asociada al índice y la TSS de su handler con los valores correspondientes. Finalmente llama a la función `copiarCódigo` que se encarga de copiar en las páginas previamente reservadas el código correspondiente a la tarea que se quiere crear según su jugador y su tipo.

2.6. Scheduler

El siguiente paso realizado por el Kernel es inicializar el scheduler. Éste es el encargado de conmutar las tareas al caer una interrupción del reloj. Para poder poner esta unidad en funcionamiento definimos las siguientes variables y arreglos globales. El arreglo scheduler de tamaño 6, presenta las tareas que se encuentran corriendo; si la i-ésima posición contiene un 1 entonces la tarea asociada al índice i está en el scheduler, caso contrario la tarea no está corriendo. La información contenida en el arreglo slots_disponibles es similar a la del arreglo scheduler, aunque con la diferencia de que en lugar de tener un 1 si la tarea i-ésima está corriendo, tiene el tipo de la tarea mencionada. Por esta diferencia y por una cuestión de declarativdad de los nombres, decidimos conservar los dos arreglos. La variable global ”actual” indica cuál es la tarea en curso de ejecución, si el valor de actual es igual a 6 entonces la tarea Idle está ejecutándose debido a que se terminó de recorrer el scheduler. Si no, contiene el índice de la tarea correspondiente en el arreglo scheduler. Es decir, por más que la tarea no se encuentre en ejecución y se esté corriendo la Idle en su lugar, esta variable contiene el índice de la tarea que debería estar corriendo, por lo que aumenta de 1 en 1. El arreglo tss_selectores contiene los selectores de segmento en la GDT que contienen los descriptores de TSS de las tareas. Los arreglos pos_x_tasks y pos_y_tasks contienen los valores de las posiciones de las tareas, que en el contexto del juego son encarnadas por pelotas. Finalmente, el arreglo acciones contiene la última acción informada por el handler de cada una de las 6 tareas. Si las tareas no se están ejecutando, el valor de la posición y la acción de la tarea pueden contener basura ya que no serán consultados.

2.6.1. sched_init

Para inicializar sus estructuras el Kernel llama a `sched_init` en el archivo sched.c. Esta función inicializa los arreglos mencionados anteriormente; scheduler, pos_x_tasks y pos_y_tasks comienzan en 0 y se completa tss_selectores con los selectores de TSS. La tarea Idle es la única tarea que comienza corriendo, por lo que la variable actual se setea en 6.

2.6.2. sched_nextTask

El archivo sched.c también provee la función que permite decidir cuál es la siguiente tarea a ser ejecutada: `sched_nextTask` que devuelve el selector de TSS de la próxima tarea. Esta función decide cuál es la próxima tarea en ejecutarse gracias a la variable actual y al arreglo scheduler. De esta forma, elige la siguiente tarea siguiendo el orden determinado por el arreglo scheduler cuyo valor contenido en este arreglo sea igual a 1. En caso de ser 0, corre la Idle, aunque la variable ”actual” funciona como ya se mencionó antes. Si la tarea que corre no es la Idle, entonces se fija si el handler de la tarea se encontraba seteado, y en ese caso se actualiza la información contenida en su TSS ya que resulta necesario actualizar su eip. De esta forma el scheduler administra las tareas y se encarga de que todas tengan tiempo de ejecución. Además, si la tarea que debería comenzar a correr tiene seteado su handler, se resetea la TSS del mismo, poniendo en `eip` el valor de la función que devuelve `setHandler`, lo que es explicado en detalle en la sección 2.7.4.

2.7. Interrupciones

Para poder trabajar con interrupciones vamos a necesitar inicializar una IDT y cargar con un puntero a ésta el Interruption Descriptor Table Register (IDTR). Las primeras 32 entradas de la IDT son entradas reservadas, la entrada 32 está ocupada por la interrupción del reloj, la entrada 33 contiene la interrupción del teclado y la entrada 47 la interrupción de las syscalls. Para inicializar la estructura de IDT es necesario llamar desde el Kernel a la función `idt_init`. Resulta necesario también configurar controlador de interrupciones, por lo que el Kernel llama a `pic_reset` y luego a `pic_enable`.

2.7.1. Rutina de atención de las excepciones

Esta rutina se encarga de llamar a la función `expecion` definida en game.c, que toma como parámetro el número de expeción que se produjo. Esta función será explicada más adelante junto con el Modo Debug en la sección 2.9.

2.7.2. Rutina de atención del reloj

Dicha rutina se encarga de numerosas tareas. En primer lugar, se fija si el juego está en pausa debido al Modo Debug, llamando a la función `estaEnPausa` (definida en game.c) que devuelve la variable ”modo_debug” explicada en la sección 2.9, es decir, devuelve un 2 en ese caso. Si el juego está efectivamente pausado, la rutina del clock finaliza y si no continúa con lo siguiente. El paso que sigue es ver si el juego finalizó, es decir, si no hay pelotas en juego y ningún jugador tiene pelotas restantes, que se chequea gracias a la función `terminoJuego`. La misma se fija lo mencionado y en caso de que el juego haya finalizado imprime en pantalla el ganador (o empate en el caso que corresponda) y pausa el juego para que no siga corriendo el clock. Esto lo hace poniendo el valor 2 a la variable global ”modo_debug”.

En el caso de que el juego esté funcionando normalmente, la rutina de reloj realiza lo siguiente. Primero llama a la función `movJugadores` que mueve los cursores de ambos jugadores según las últimas direcciones tomadas del teclado cada 30 ciclos de clock. Luego se encarga de mover las pelotas según las acciones que guardamos cuando los handlers las informaban. La función `movPelotas` es la que decide si la pelota rebota, o si es ”gol”, por lo que hay que matar la tarea correspondiente y sumar los puntos. También realiza los movimientos en pantalla y actualiza las posiciones de las pelotas en los arreglos pos_x_tasks y pos_y_tasks. Estas dos últimas funciones se encuentran en game.c. Luego se fija si el TR actual corresponde a la TSS de una tarea de tipo handler, en cuyo caso debe morir la tarea a la que se encuentra asociado. Esto se realiza con las funciones `esHandler` y `matarTarea`. La primera se encarga de devolver, si es un handler, el selector de segmento de la tss de la tarea que debe morir, y un 0 en caso contrario. De esta manera, la segunda función recibe este valor y se encarga de eliminar la tarea o, si el parámetro es 0, no hace nada. Esto se detallará más adelante en la sección 2.8. Finalmente la rutina llama a `sched_nextTask` que devuelve un selector de segmento. Este es comparado con el valor actual del TR: si es el mismo valor, finaliza la rutina y si no, se realiza un jmp far a la siguiente tarea.

2.7.3. Rutina de atención del teclado

La rutina de atención del teclado lee la tecla a través del puerto 0x60 (con la instrucción `in al, 0x60`) de la que se obtiene un scan code. Como al soltar la tecla a este código se le suma 0x80, le resta este valor. Luego se llama a la función `leerTeclado` que se encuentra en el archivo game.c, que toma como parámetro el scan code leído. Dicha función se encarga de lo siguiente: si la tecla presionada fue alguna de las que indican que la barra de un jugador debe ser movida hacia arriba o abajo, guarda esta acción en el arreglo direccionJug, en la posición correspondiente a cada jugador. Su utilidad y comportamiento serán detallados en la sección 2.8. Si la tecla indica que el jugador quiere tirar una pelota de cierto tipo, se llama a la función `crearPelota` con el jugador y el tipo de tarea. Esta función también será explicada con mayor detenimiento en la sección mencionada. Finalmente, si la tecla era la ’y’, llama a la función `modoDebug`, cuyo funcionamiento será expuesto en la sección 2.9.

Luego de llamar a esta función, se fija si la tecla era un número, en cuyo caso se debe imprimir en la esquina superior derecha según indica la consigna. Dado que el scan code del 1 es 2, y va aumentando de 1 en 1 hasta llegar al 0 (su scan code es 11), guardamos en memoria (con la etiqueta `nros`) los números en orden del 1 al 9 y luego el 0, por lo que si le restamos 2 al scan code podemos acceder al número sumándole dicho valor a la etiqueta. Entonces la rutina hace lo siguiente: le resta 2 al valor del scan code y si este es mayor a 9 o menor a 0 (es decir, no es el scan code de un número) finaliza. En caso contrario, imprime en pantalla lo que hay en [nros + scan code - 2].

2.7.4. Rutina de atención de las syscalls

Esta función lee el registro `eax`, y se fija cuál de los cuatro tipos de syscall interrumpió. Luego, salta a la correspondiente y llama a la función que se ocupa de resolverla que fueron implentadas en C, que se encuentran en el archivo *game.c*.

La primera función se llama `sys_talk` y lo que hace es imprimir el mensaje pasado por parámetro en la fila correspondiente según qué tarea produjo la interrupción. La syscall "where" fue dividida en dos funciones: `sys_where_x` y `sys_where_y`. Para realizarlas utilizamos los arreglos pos_x_tasks y pos_y_tasks: devolvemos el valor de la posición "actual", aunque en el caso de la posición x, nos fijamos de qué jugador es la tarea, ya que en caso de ser del 2, había que comenzar a contar desde la derecha. La syscall `sys_setHandler` se encarga de guardar en el arreglo "handlers" la TSS del handler correspondiente que busca en el arreglo TSS_ARRAY, luego pone el eip del mismo en el puntero a función que llega como parámetro y finalmente guarda este puntero en un arreglo denominado "fun_handlers" que tiene 6 posiciones y eventualmente contendrá los punteros a funciones de todos los handlers que sean seteados. Por último la función `sys_informAction` se encarga de guardar la acción en el arreglo ya mencionado "acciones".

2.8. Juego

El siguiente paso del Kernel es cargar la tarea inicial, luego llama a la función `mapearTareas` y a `game_init`. Después habilita las interrupciones con `sti` y hace un jmp far a la tarea Idle. Para mantener el correcto funcionamiento del juego, vimos necesaria la utilización de varias estructuras y variables auxiliares:

- direccionJug: tenemos un array de dos elementos del tipo `enum e_action` para actualizar, en su debido momento, las posiciones de los jugadores.
- contador: variable global utilizada para indicar el momento en el que se deben actualizar las posiciones de los jugadores, en nuestro caso, a los 30 ciclos de reloj.
- pos_jugadores: array encargado de almacenar las posiciones de los jugadores A y B.
- goles: es responsable de almacenar el puntaje de cada participante.
- pelotas_restantes: arreglo que guarda la cantidad de pelotas que le restan por utilizar a cada jugador, información que debe ser printeada por pantalla.
- fun_handlers: contiene el puntero a la función que tiene que correr el handler de una tarea, éste se asigna cuando la tarea setea su handler.
- acciones: contiene en la i-ésima posición la acción que está realizando la pelota correspondiente, puede ser 'Up','Down' o 'Center'.
- inversa_y: contiene un 0 en la i-ésima posición si la pelota correspondiente rebotó una cantidad par de veces contra el borde superior o inferior, y un 1 si rebotó una cantidad impar de veces.
- inversa_x: contiene un 0 en la i-ésima posición si la pelota correspondiente rebotó una cantidad par de veces contra los extremos laterales, y un 1 si rebotó una cantidad impar de veces.

- cruces_pelotas: posiciones en x de las 'cajitas' que contendran una cruz si el slot con el que se corresponden se encuentra ocupado y un espacio si se encuentra disponible. Elegimos guardar esta información en un arreglo para facilitar la impresión en pantalla al crear una pelota o al matarla, y de esta forma actualizar la pantalla.
- modo_debug: variable global encargada de regular el funcionamiento del modo debug, explicada más adelante.
- text_excep: arreglo que contiene el nombre de todas las excepciones como `char*`, las utilizaremos en el modo debug.

En los arreglos acciones, inversa_y y inversa_x si la i-ésima pelota no se encuentra en el scheduler, la posición correspondiente contiene basura.

2.8.1. game_init

Esta función se encarga de inicializar los arreglos mencionados anteriormente así como las variables con los valores correspondientes. Además imprime la pantalla de inicio del juego.

2.8.2. movJugadores

Función encargada de mover a las barras de los jugadores, lo cual debe realizarse cada 30 ciclos de clock. Por esta razón, debemos chequear si el contador llegó al mismo número. En caso de que esto no sea correcto, lo incrementamos en uno, sino lo reiniciamos y pasamos a efectuar los movimientos. Por cada jugador, observamos las variables del array direccionesJug para determinar si debemos rellenar un posición por encima de la barra y despintar una por debajo de la barra (Up), o realizar los prints en el sentido opuesto (Down), o no hacer nada (Center). Previamente, debemos chequear si el movimiento es válido. Además, debemos actualizar las posiciones de las barras (en el arreglo pos_jugadores) en caso de que se muevan, y las direcciones de ambos jugadores a Center.

2.8.3. movPelotas

Como esta función se llama en cada ciclo de reloj, lo primero que hace es fijarse si es momento de mover las pelotas, es decir si es el fin de la vuelta del scheduler. Esto se traduce en nuestra implementación en chequear si la variable actual tiene el valor 6. En ese caso recorre todas las tareas del scheduler haciendo lo siguiente: se fija la dirección de la pelota y los casos en los que debe rebotar, cambiando inversa_y y guardando la nueva posición en y en una variable. Luego, en otra variable guarda el valor de x correspondiente según si estaba inversa_x o no. Con este nuevo valor chequea si la pelota rebota contra alguna de las barras de los jugadores o si fue gol. A continuación detallamos los casos en los que decidimos si fue o no gol, dependiendo de si la pelota estaba dirigida hacia arriba o hacia abajo.

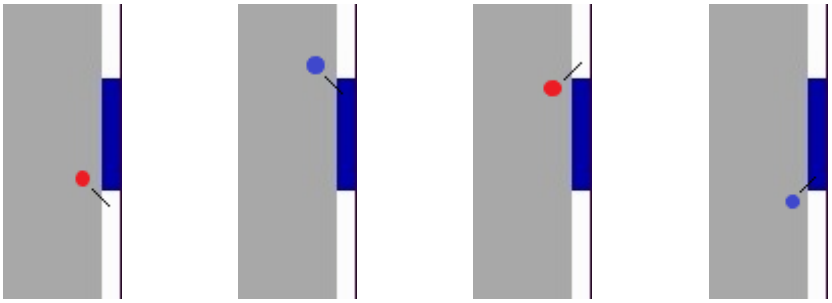

Figura 1: Definimos los casos del 1 al 4, de izquierda a derecha

En la figura 1 vemos el caso 1, la pelota viene dirigida hacia abajo, es decir su acción es 'Down'. Al actualizar las coordenadas de la pelota, éstas no coincidirán con la posición en la que se encuentra el jugador, sin embargo consideramos que la pelota está rebotando y por lo tanto no es un gol. Por este motivo, al realizar el chequeo de posiciones comparamos la coordenada y de la pelota con el límite inferior del jugador incrementado en uno. Al contrario, en el caso 2, al actualizar la posición de la pelota, ésta coincidirá con la posición del jugador pero asumimos que sí se realizó un gol ya que la pelota no tiene superficie para rebotar. Por esto comparamos la posición de la pelota actualizada con la del límite superior del jugador al cual le restamos uno al comparar con un menor estricto. De forma análoga resolvimos los casos 3 y 4 cuando la pelota está dirigida hacia arriba.

Si se determina a partir de las posiciones de la pelota y del jugador en cuestión que se produjo un gol se llama a la función `matarTarea`, se actualiza el arreglo que contiene los puntajes y se imprime la actualización de éstos.

Si no fue gol, se imprime la pelota en la nueva posición con las variables en las que se guardaron los nuevos valores de x e y. Luego se borra del lugar anterior con los arreglos pos_x_tasks y pos_y_tasks y finalmente se actualizan los valores de los mismos.

2.8.4. matarTarea

Toma un parámetro, un entero con signo, el cual debe indicar el numero de la tarea para poder desalojarla. Si el parámetro es -1, no se realiza ninguna modificación. Para desalojar una tarea, seteamos en cero sus posiciones en los arrays scheduler, slots_disponibles y handler, para no tenerlas más en cuenta. Luego, eliminamos la cruz que estaba marcada en su respectiva caja en pantalla, también el mensaje que había enviado la pelota y la eliminamos del tablero usando pos_x_tasks y pos_y_tasks.

2.8.5. crearPelota

Esta función es llamada por la función `leerTeclado` cada vez que uno de los jugadores quiere lanzar una nueva pelota. Por este motivo, el primer paso es confirmar si efectivamente el jugador puede realizar lo solicitado. Esto lo hace fijándose, con el arreglo pelotas_restantes, si el jugador no lanzó todavía sus 15 pelotas, y en ese caso se chequea que no tenga 3 pelotas en juego. Esto lo hace recorriendo las posiciones correspondientes al jugador en slots_disponibles. Si todas las condiciones mencionadas se cumplen llama a la función `tss_task_init`, le resta 1 a las pelotas restantes del jugador, pone inversa_y, inversa_x, pos_x_tasks y pos_y_tasks con los valores correspondientes, la inicializa en el scheduler, printea una 'X' en la "cajita" correspondiente y finalmente imprime la pelota en la posición calculada (empieza desde el medio de la barra del jugador que la creó).

2.9. Modo Debug

Para implementar el Modo Debug creamos en primer lugar una variable global llamada "modo_debug". Si su valor es 0, el Modo Debug está desactivado, si es 1 está activado pero el juego está en marcha, y si es 2, el juego está pausado. A continuación detallaremos las funciones que utilizamos para esta funcionalidad del juego.

2.9.1. modoDebug

Como ya mencionamos en la rutina de interrupción del teclado, al apretar la tecla 'y' se llama a esta función. Si la variable modo_debug estaba en 0, se cambia a 1 y viceversa. Si la variable estaba en 2, entonces significa que se había producido una excepción y se había impreso en pantalla según indicaba la consigna, por lo que en este caso no solo se cambia el valor de la variable a 1 sino que también se vuelve a pintar la pantalla como estaba antes de la excepción. Esto lo hace pintando todo de gris de nuevo y fijándose dónde se encontraban las pelotas recorriendo el scheduler, pos_x_tasks y pos_y_tasks.

2.9.2. excepcion

Esta función es llamada cada vez que se genera una excepción por lo que primero chequea que la variable modo_debug esté en 1. En ese caso le cambia el valor a 2 y dibuja en pantalla el recuadro como está indicado en la consigna, utilizando el arreglo ”text_excep”, indexando con el número de expeción. Los valores de los registros los saca de la TSS de la tarea que estaba corriendo. Esta TSS se obtiene haciendo `TSS_ARRAY[1 + actual]` que devuelve un puntero a la misma. Por último mata la tarea actual llamando a `matarTarea` con la variable ”actual”.