DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

System programming Pong

Organización del Computador II

Integrante	LU	Correo electrónico
Bruno Gomez	428/18	brunolml99@outlook.es

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

A continuación describiremos en cada sección las decisiones tomadas y los detalles de la implementación de las estructuras y módulos kernel correspondientes.

2. General Descriptor Table (GDT)

En la GDT las entradas 14 y 15 corresponden a los descriptores de segmento de código de kernel y código de usuario. Las entradas 16 y 17 corresponden a los descriptores de segmento de los datos de kernel y los de usuario respectivamente. Los descriptores que pertenecen al kernel tienen su campo DPL en 0, pues poseen el privilegio más alto. Sin embargo para los descriptores de código y datos de usuario el DPL es 3. En el caso de estos 4 descriptores se encuentra seteado el bit de granularidad que nos permite extender el alcance del limite para poder direccionar los 163 MB de memoria. Como empleamos un esquema de segmentacion Flat, todas las bases de estos descriptores de segmento están en cero.

La entrada 18 corresponde al descriptor de segmento de pantalla. El campo base de este descriptor apunta a la dirección indicada por el enunciado para el segmento de vídeo.

Las entradas 19 y 20 corresponden al descriptor de Task State Segment (TSS) de la tarea inicial y la tarea Idle respectivamente. Luego en las entradas 21 a 26 tenemos los descriptores de TSS de las tareas de cada jugador, primero las 3 tareas del jugador A y luego las 3 tareas del jugador B. Finalmente desde la entrada 27 a la 32 tenemos los descriptores de TSS de los handlers correspondiente a cada tarea. Para los descriptores de TSS el DPL es cero, pues queremos que para poder conmutar las tareas se posea privilegio supervisor. Además su bit de sistema está en 0, y sus bits de tipo son los adecuados para un descriptor de TSS. Para mayor claridad en ”defines.h” agregamos el reemplazo sintáctico de los índices para tener una mejor legibilidad del código.

3. Interrupt Descriptor Table (IDT)

En la IDT inicializamos las 20 entradas correspondientes a los descriptores de las excepciones predefinidas del procesador. Los descriptores de estas excepciones tienen su DPL en cero por ser interrupciones que son generadas por el procesador. Las entradas 32 y 33 corresponden a los descriptores de las interrupciones de hardware. El DPL de estos descriptores es cero. La entrada 47 es para el descriptor de la interrupción para los servicios del sistema, dado que son interrupciones de software que van a ser llamadas desde las tareas, el DPL de este descriptor es 3. Para todos los descriptores de interrupciones que definimos en la IDT su bit de sistema se encuentra en cero, su tipo corresponde a una puerta de interrupción, además el selector de segmento que contienen estos descriptores corresponde al definido en la GDT para el segmento de código del kernel.

4. Memory Management Unit (MMU)

Utilizamos una estructura donde almacenamos las direcciones de las páginas físicas en las que vamos a copiar los códigos de las tareas, donde se almacenarán las pilas de nivel cero de las tareas y los handlers, y las páginas donde se van a crearan los directorios de cada tarea. Estas páginas son estáticas, son pedidas en el arranque del kernel y durante el juego sólo se cambia su contenido. En cada uno de estos arreglos el índice representa a cada una de las 6 tareas que pueden estar corriendo simultáneamente en el juego. Es decir si accedemos al primer elemento de tareas encontramos la pagina de código correspondiente a la primer tarea. Si accedemos al primer elemento de SS0_tareas encontramos la pila de nivel cero de la misma tarea y así para todos los arreglos.

```
typedef struct {
    uint32_t tareas[6];
    uint32_t SS0_tareas[6];
    uint32_t SS0_handlers[6];
    uint32_t directorios[6];
```

```
} Mapa_paginas_fisicas_t;
```

En la función mmu_initTaskDir(), recibimos como parámetro el índice en el arreglo ”directorios”, que indica que tarea se va a mapear. Apuntamos el campo base de la primer entrada de la tabla de directorios a la Tabla de Pagina que usa el Kernel para mapear su espacio de direcciones y así tener mapeado el mismo con identity mapping en cada tarea sin tener que crear una nueva tabla. En la segunda entrada del directorio apuntamos a una nueva Tabla de página donde mapeamos las dos páginas correspondientes al código de la tarea (siendo la primera página la apuntada por el arreglo tareas). Este mapeo se realiza en el arranque del kernel para cada una de las seis tareas porque no va a cambiar durante el desarrollo del juego.

5. Task State Segment (TSS)

En este módulo del kernel se incializan los tss para la tarea inicial, la tarea idle y las 6 tareas de los usuarios, y se completa la base de las entradas de la gdt que los describen. Se completan en los tss los valores estáticos y los valores que se pueden conocer en tiempo de compilación. En nuestro caso, como los directorios de las tareas son estáticos, esto incluye al cr3. Adicionalmente, este módulo contiene las funciones ïnit_task_tss.ºinit_handler_tss”que se encargan de inicializar los valores dinámicos de los tss de las tareas de usuario y de sus handlers respectivamente.

6. Scheduler

Diseñamos una estructura llamada scheduler que va a contener todo lo necesario para poder administrar el cambio de tareas en cada ciclo. Para lograr esto la estructura cuenta con el campo current_task, que se utiliza para saber cual es la tarea que se encuentra corriendo, y también un flag is_current_handler, que dice si lo que esta corriendo actualmente es el handler de la tarea actual o la tarea propiamente dicha. Por otro lado, se tiene una variable ticks, que se encargara de indicar la cantidad de ticks que han pasado, para saber cuando hay que actualizar las posiciones de las pelotas en juego y las paletas de los jugadores.

```
typedef struct{
        uint8_t current_task;
        uint8_t is_current_handler;
        uint8_t debug_mode;
        uint8_t ticks;
        uint16_t task_selectors[6];
        uint16_t handler_selectors[6];
        f_handler_t handler_ptr[6];
        uint8_t on_interrupt;
} Scheduler;
```

En cada interrupción de reloj se llama a la función ”sched_nextTask” que chequea si hubo una interrupción del procesador (”scheduler.on_interrupt”) y si el modo debug está activado (”scheduler.debug_mode”), de ser así se devuelve la tarea idle y no se actualiza el número de tics. De esta forma, el cuadro con la información de debug permanece en pantalla hasta que se presione la tecla z”para limpiar el valor de ”scheduler.on_interrupt”.

Si estas condiciones no se cumplen, se procede a ver si la próxima tarea seteó su handler verficando si hay un puntero almacenado en el índice correspondiente de ”scheduler.handler_ptr”. Si esto se cumple, se devuelve el selector del handler y no se incrementan los ticks. Al no incrementar los ticks, cuando se vuelva a llamar a ”sched_nextTask” luego de que el handler informe la acción se devolverá el selector de la tarea actual. En este caso, se incrementan los ticks para pasar a la siguiente tarea en el siguiente llamado.

Cuando una tarea no fue lanzada, su selector apunta a la tarea idle. Luego de devolver las seis tareas posibles y sus handlers en el caso en que estuvieran seteados, la función ”sched_nextTask”llama a las

funciones que actualizan las estructuras del juego y dibujan la pantalla, y devuelve la tarea idle para posteriormente volver a ciclar por las tareas de los jugadores.

7. Rutinas de Atención de interrupciones

- Teclado (isr33): El código en assembler utiliza una función auxiliar escrita en C. La cual según la tecla presionada llama a otra función que maneja cada caso. En el caso de las teclas que implican el lanzamiento de una tarea . Se busca una posición libre en la estructura del kernel que maneja las tareas, si existe se procede a inicializar su tss con los valores dinamicos utilizando la función tss_initTask descripta en la sección TSS y se carga este selector en la estructura que utiliza el scheduler para conmutar tareas. Adicionalmente se setean los valores iniciales que requiere el juego para dibujar la pelota.
- Reloj (isr32) : Esta interrupción comienza chequeando si la tarea que se interrumpió fue un handler, porque si este fuera el caso la tarea es terminada. Luego se llama a la función del scheduler que provee el selector de tss de la siguiente tarea. Se compara si no es el de la tarea actual y en dicho caso se salta a la nueva tarea.
- Excepciones Predefinidas : Para el caso de las excepciones predefinidas, se manejan con una única rutina. Realiza los chequeos necesarios para el funcionamiento del modo debug con funciones escritas en C. Y se contempla el caso en el que una tarea realice una excepción es terminada y en su lugar se corre la tarea Idle.

8. Servicios del sistema

- InformAction: Si llega una accion invalida se mata a la tarea actual y se corre la tarea Idle. Sino se registra en una estructura del juego la accion de la pelota. Esta estructura sera consultada al momento de dibujar la pantalla en cada iteración del juego.
- Where: Dejamos un espacio en el stack para poder devolver el valor de la posicion en este. Luego preservamos todos los registros y le pasamos un puntero al espacio que dejamos en el stack a una función. Esta función utiliza estructuras de game para poder calcular la posición de la pelota relativa a su jugador y las escribe en el puntero dado. Luego popeamos los registros que preservamos y realizamos pop sobre los registros que debemos volver la posición, aprovechando que dejamos los valores en el stack.
- Talk: Llamamos a una función escrita en C a la cual le pasamos el puntero al mensaje. Esta función copia el mensaje dentro del rango permitido y lo almacena en una de las estructuras de game.
- SetHandler: Se llama a una función desde el servicio. Esta almacena el puntero en la estructura del scheduler que posee los handlers de las tareas.

9. Game

9.1. Estructuras auxiliares

Se utilizara esta estructura de coordenadas para subsiguientes estructuras donde se necesiten guardar coordenadas de objetos.

```
typedef struct {
        int8_t x;
        int8_t y;
} Point;
```

Este enum se utilizara para simbolizar las 2 posibles direcciones en las que puede ir una pelota.

```
typedef enum {
        DIR_LEFT = 0,
        DIR_RIGHT
} Direction;
```

Este typedef se utilizara para definir a las ”paddle”de los jugadores, es decir, las barras que se mueven para arriba o abajo dependiendo que tecla se presione por el jugador correspondiente.

```
typedef Point Paddle;
```

Utilizaremos este typedef para definir las posibles acciones de las ”paddles.º barras de los jugadores.

```
typedef enum{
        MOVE_PADDLE_UP = 0,
        MOVE_PADDLE_DOWN = 1,
        PADDLE_STILL = 2
} paddle_action_t;
```

9.2. Estructuras principales

Esta estructura se encargara de mantener el estado de una bola que este corriendo en el juego. Tendremos un flag alive que indicara si la pelota esta en el juego o no. Se almacenara la posicion de la pelota, asi tambien como su direccion, y tambien un flag inverted_action que servira para saber si las acciones de la pelota deben ser invertidas o no.

```
typedef struct {
        uint8_t alive;
        Point position;
        Direction direction;
        uint8_t inverted_action;
}Ball;
```

Definimos una estructura de datos para un jugador. En esta se encuentra un arreglo de pelotas, donde se almacena toda la informacion necesaria de una pelota que se menciono en la estructura anterior, siendo MAX_BALLS la maxima cantidad de pelotas que pueden haber en juego, osea 3. Luego se tienen para cada pelota el tipo de tarea que la esta manejando y el mensaje que la tarea que representa a esa pelota emitio a traves del syscall talk. Tambien se guarda la accion que la tarea que representa a esa pelota informo a traves del syscall blueinformAction en balls_actions, y finalmente se guarda, en caso de que el jugador haya presionado una tecla para crear una nueva tarea, el tipo de tarea asociado a esa tecla en el arreglo new_ball_type. Por otro lado, se tiene tambien la variable paddle que almacenara la posicion de la tablita del jugador y una variable balls_left para indicar cuantas pelotas mas puede utilizar el jugador en el juego. Finalmente el campo score almacenara el puntaje del jugador y paddle_action contendra la ultima accion indicada por el jugador a traves del teclado para mover a la tablita del mismo.

```
typedef struct{
        Ball balls[MAX_BALLS];
        uint8_t balls_type[MAX_BALLS];
        char* balls_msg[MAX_BALLS];
        e_action_t balls_actions[MAX_BALLS];
        Paddle paddle;
        uint8_t balls_left;
```

```
        uint8_t score;
        paddle_action_t paddle_action;
        uint8_t new_ball_type[MAX_BALLS];
}Player;
```

Esta estructura a diferencia de las anteriores se utilizara para almacenar las coordenadas de los datos que deben mostrarse en el panel inferior del juego para un jugador, ademas de algunos colores y tamaños necesarios para graficarlo. Para esto se tiene la variable location que contendra las coordenadas donde se encuentra situado el tope izquierdo del panel, luego su ancho y alto estan en los campos width y height. Tambien se tiene el color de fondo del panel en bg_color, y el color de frente de los indicadores de tareas libres en task_slot_color. Se tienen por otro lado guardadas las coordenadas del indicador de puntaje en score, de las pelotas restantes en balls_left y de los que indican que tareas se estan corriendo en el arreglo active_task_type. Finalmente en el arreglo active_task_msg se tienen las coordenadas para situar los mensajes de las tareas que manejan a cada pelota y en el arreglo task_slot se tienen las coordenadas para indicar si un slot esta libre para lanzar una nueva pelota.

```
typedef struct{
        Point location;
        uint8_t width;
        uint8_t height;
        uint8_t bg_color;
        uint8_t task_slot_color;
        Point score;
        Point balls_left;
        Point active_task_type[MAX_BALLS];
        Point active_task_msg[MAX_BALLS];
        Point task_slot[MAX_BALLS];
}InfoPanel;
```

9.3. Funcionamiento

Cuando arranca el sistema se llama a la función ”game_init .ésta se encarga de inicializar todas las estructuras mencionadas anteriormente. Primero se inicializa el estado del juego seteando los valores de los jugadores en sus coordenadas iniciales, la cantidad de pelotas que va a tener cada jugador, el puntaje en cero y luego todas las pelotas se las inicializa ” muertas”. Luego se setea el tablero inicial con los colores correspondientes indicados en el enunciado y se dibujan los jugadores en sus posiciones iniciales. Junto con estos se inicializa el panel que presenta la información sobre cada jugador y sus tareas luego de haber hecho todo esto se dibuja todo en pantalla.

Cada vez que se tiene que actualizar el juego se realizan los siguientes pasos. Primero se actualizan las paletas de los jugadores, chequeando la acción que se guardo en la estructura de jugadores .Esta acción se actualiza cada vez que hay una interrupción de teclado, guardando el valor correspondiente para cada tecla. Luego de actualizar las posiciones de los jugadores se setea la acción de la paleta al valor equivalente a ”no moverse ”. A continuación se actualizan las pelotas de los jugadores. Para cada pelota viva se chequea la acción que fue informada por la syscall ”InformAction”por parte de la tarea. Si la tarea en cuestión no tiene un handler registrado la accion por default va a ser quedarse en el centro. Tomando en cuenta la posición, el sentido y la flag ”inverted_action ”, la cual nos dice que la accion debe ser invertida, realizamos el calculo para la siguiente posición de la pelota. En el caso de que la próxima posición caiga dentro de una de las zonas de ”goal ”se mata a esa pelota y se incrementa el puntaje del jugador correspondiente. Finalmente se actualiza la parte gráfica del juego, borrando primero las posiciones anteriores de las los jugadores y las pelotas, utilizando el estado anterior que se mencionó previamente y luego se dibujan las posiciones nuevas. En el caso en el que ningún jugador tenga pelotas para poder correr, quiere decir que el juego terminó. Se imprime en pantalla el jugador que ganó y de actualizar el juego.

Todo el manejo de la parte gráfica mencionada anteriormente, escribe sobre un buffer que representa la gráfica del juego. Esto último se utiliza para actualizar las posiciones del buffer de vídeo que difieren

con las de buffer del juego.

Cuando se actualizaron los jugadores y las pelotas se guarda el estado previo de cada uno para posteriormente dibujarlo en pantalla eficientemente.

9.4. Comportamiento de las tareas

Dejamos a disposición las tareas default con modificaciones para poder testear distintos aspectos con los que debía cumplir el sistema. La tarea taskA1 se encarga de utilizar el syscall talk para imprimir sus coordenadas que vienen por el syscall where. Por otro lado, el taskA2 nunca realiza el llamado a informAction en su handler, entonces debe ser desalojado. La tarea taskB1 intenta escribir en direcciones lineales que no le corresponden a su espacio de memoria, entonces genera una excepción y debe ser desalojada también. También el taskB3 intenta setear su handler 2 veces, entonces también debe ser matada, y finalmente la tarea taskB2 imprime sus coordenadas como la tarea taskA1, solo que en este caso sirve para chequear si las coordenadas son relativas al jugador que pertenecen.