Organización del Computador II
TP2

16 de julio de 2015

Integrante	LU	Correo electrónico
Federico Beuter	827/13	federicobeuter@gmail.com
Juan Rinaudo	864/13	jangamesdev@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

La idea de este trabajo es poder aplicar los conocimentos de *System Programming* adquiridos en clase. Para lograr esto se implemento un pequeño *kernel* para procesdores x86, el mismo maneja un entorno en donde pueden correr hasta 16 tareas simultaneas, la responsabilidad del *kernel* es regular la ejecucion de estas tareas valiendose de mecanismos de proteccion provistos por el procesador.

2. Ejercicio 1

Para la resolucion de este ejercicio modificamos `kernel.asm`, `gdt.c`, `gdt.h`, `defines.h` y `screen.c`.

2.1. a

Comenzamos desde `C` creando los defines de los indices de los descriptores de segmento en la gdt en `defines.h`, como los primeros 8 indices se consideran utilizados vamos a enumerar nustros descriptores desde 8 (Como indica el subindice a).

- `GDT_NIVEL0_CODIGO 8`
- `GDT_NIVEL0_DATOS 9`
- `GDT_NIVEL3_CODIGO 10`
- `GDT_NIVEL3_DATOS 11`

Luego en el archivo `gdt.c` agregamos al array `gdt` los 4 descriptores, con su respectivo nivel (0 para los descriptores del kernel y 3 para los de usuario) y tipo (`0xA` para los de codigo y `0x2` para los de datos), su base la posicionamos al principio de la memoria, con un limite de 500MB seteando granularity (`G = 1`) y el valor de limite (`0x1F400`), marcandolo como presente (`P = 1`) y de 32 bits (`D/B = 1`) y no de sistema (`S = 1`).

2.2. b

Para este subindice pasamos a `kernel.asm` donde habilitamos **A20** (utilizando la funcion brindada por la catedra `habilitar_A20`) y luego cargamos la **GDT** usando `LGDT` y un puntero al descriptor de **GDT** `GDT_DESC` (Estructura brindada por la catedra y definida en `gdt.h`). A continuacion habilitamos la proteccion (`PE = 1`. Protected Mode Enable) en **CR0** pasandolo a `EAX` haciendo un `OR` y moviendolo nuevamente a `CR0`. Ahora para pasar a modo protegido hacemos un salto al descriptor de nivel 0 de codigo (`0x40`) en la **GDT** a travez de un `JMP FAR`.

Una vez en modo protegido seteamos los selectores de segmento de datos y de stack de nivel 0, y finalizamos posicionando la pila (`EBP` y `ESP`) en `0x27000`.

2.3. c

Para este punto volvemos a `defines.h` y definimos un quinto indice para el descriptor de la memoria de la pantalla. `GDT_PANTALLA 12`

Luego en `gdt.c` agregamos al array `gdt` un descriptor que comienza en `0xB8000` (Memoria de video) con un limite de 4KB, de nivel 0, con granularidad y el resto de los bits igual que los descriptores anteriores.

2.4. d

Para este punto vamos a `screen.c` y completamos las siguientes funciones auxiliares:

- `screen_inicializar`: Limpia la pantalla y luego llama a `screen_pintar_puntajes`.
- `screen_pintar_rect`: Pinta un rectangulo de un color en la pantalla, comienza de un punto (x, y) y tiene un tamaño relativo (width, height).
- `screen_pintar_puntajes`: Pinta los puntajes igual que la figura 3 del enunciado.

3. Ejercicio 2

Para la resolucion de este ejercicio modificamos `idt.h`, `idt.c` y `isr.h`, `isr.asm`.

3.1. a

Para manejar las entradas de la **IDT** la catedra nos proveyo de la estructuras de datos necesarias. Para todas las entradas los atributos fueron los siguientes:

- `offset`: `&_isr??`, donde `??` es el numero de interrupcion.
- `selseg`: `0x40`, el segmento de codigo de nivel 0.
- `attr`: `0xEE0` para la interrupcion `0x46` i `0x8E0` para las demas.

A todas las entradas se las marco como presente (`P = 1`), se les puso el privilegio adecuado (`DPL = 0` y `DPL = 3` para que sea llamada por el usuario) y se seteo a las interrupciones como *interrupt gate* (`GATE TYPE = 14`.

A cada uno de los handlers se los definio en `isr.h`, y se cargaron todas las entradas de la **IDT** en la funcion `idt_inicializar`.

El codigo de cada *handler* se encuentra en el fichero `isr.asm`, en esta etapa el mismo consiste en imprimir el numero de interrupcion por pantalla.

3.2. b

Para cargar la **IDT** alcanza con utilizar la funcion `LIDT` con el puntero a memoria `IDT_DESC`, definido por la catedra. Previo a esto hay que llamar a `idt_incializar` para cargar las entradas.

4. Ejercicio 3

Para resolver este ejercicio se modicara `screen.c`, `screen.h`, `mmu.c`, `mmu.h`, `kernel.asm` y `defines.h`.

4.1. a

Para comenzar vamos a modificar `screen.c` y `screen.h`, creamos la funcion en `C` `screen_refrescar` que se encargara de limpiar la pantalla y dibujar el fondo usando `screen_inicializar` y luego escribir la informacion del juego (Los datos de las tareas de ambos jugadores y los puntajes) usando la funcion `print` (Brindada por la catedra).

4.2. b

Definimos `DIR_PAGINAS_KERNEL` como `0x27000` en `defines.h`, luego vamos a `mmu.h` y creamos `mmu_inicializar_dir_kernel`, esta funcion crea en la posicion de memoria definida anteriormente el directorio de tablas de paginas del kernel y en la siguiente pagina (`0x28000`) la primera tabla de paginas del kernel, la memoria sera mapeada con identity mapping de `0x0` a `0x3FFFFF`.

4.3. c

Vamos a `kernel.asm`, llamamos a `inicializar_mmu` y despues a `mmu_inicializar_dir_kernel` para crear el directorio de tablas y las tablas. Luego muevo a **CR3** la posicion del directorio de tablas de paginas (`DIR_PAGINAS_KERNEL`). Luego activamos paginacion en **CR0** (**PG = 1**) (Paging) de **CR0**.

4.4. d

Para terminar en `screen.c/h` creamos la funcion `print_group` que usa print para escribir el nombre del grupo (Alineado a la derecha, tomando el tamaño del string y restandoselo al tamaño total de la pantalla) y la llamamos desde `kernel.asm`.

5. Ejercicio 4

Para resolver este ejercicio se modicara `mmu.c`, `mmu.h` y `kernel.asm`.

5.1. a

Comenzamos editando `mmu.c` y creando la funcion `mmu_inicializar` que se asigna la primera pagina de la memoria libre para guardar valores del sistema. Y en `0x100000` guarda un puntero a la siguiente pagina util.

5.2. b

Ahora escribimos `mmu_inicializar_dir_pirata` que se encargara de crear el directorio de pagina de las tareas, que pide dos paginas libres y crea el directorio de tablas en la primera y una tabla de paginas en la segunda, haciendo identity mapping de los primeros 500MB. Y luego mapeamos un sector de la memoria que no esta asignado a nada (`0x401000`) a la posicion del mapa correspondiente (`X:1,Y:1` si es el primer jugado y `X:78,Y:48` para el segundo jugador), copia el codigo de la tarea (Que se pasa como parametro a la funcion) para terminar desmapeamos la posicion mapeada anteriormente.

5.3. c

Para la tarea anterior es neceario crear las dos funcciones `mmu_mapear_pagina` y `mmu_unmapear_pagina` que con un **CR3**, una direccion virtual y una fisica en el caso de mapeo o una direccion virtual en el caso de desmapeo modifica el directorio de tablas y la tabla de paginas de la **CR3** pasada como parametro y mapea o desmapea la pagina.

5.4. d

Para este punto vamos a `kernel.asm` y creamos un segmento de codigo para testear las cosas echas. Llamamos a `mmu_inicializar_dir_pirata` y cambiamos el **CR3** del sistema con el que devuelve la funcion. Modificamos la memoria de video para cambiar el color de fondo del primer caracter de la pantalla. Esto luego fue eliminado del kernel.asm como pide el subindice.

```
1    PUSH <POS_CODIGO>
2    MOV EAX, CR3
3    PUSH EAX
4    PUSH <POS_PIRATA>
5    CALL mmu_inicializar_dir_pirata
6    SUB ESP, 12
7    MOV CR3, EAX
8    CALL print_group
```

6. Ejercicio 5

Para la resolucion de este ejercicio modificamos `isr.asm` y `game.c`.

6.1. a

Estas entradas ya fueron inicializadas en el ejercicio 2.

6.2. b

```
1    PUSHAD
2    CALL game_tick
3    POPAD
4    IRET
```

Luego, la funcion `game_tick` fue modificada para llamar a `screen_actualizar_reloj_global` tal como pide el enunciado.

6.3. c

```
1    PUSHAD
2    CALL fin_intr_pic1
3    XOR eax, eax
4    IN al, 0x60
5    MOV ecx, 0x000F000F ; color
6    PUSH ecx
7    MOV ecx, 0          ; poscion
8    PUSH ecx
9    MOV ecx, 0          ; posicion
10   PUSH ecx
11   PUSH eax            ; codigo tecla
12   CALL print
13   ADD esp, 16
14   POPAD
15   IRET
```

6.4. d

```
1    PUSHAD
2    MOV eax, 0x42
3    POP edi              ; popeo todo menos eax
4    POP esi
5    POP ebp
6    ADD esp, 4
7    POP ebx
8    POP edx
9    POP ecx
10   ADD esp, 4
11   PUSH eax
12   POP eax
13   IRET
```

7. Ejercicio 6

Para resolver este ejercicio se modicara `tss.h`, `tss.c`, `kernel.asm` y `defines.h`.

7.1. a

Primero vamos a `defines.h` y definimos los indices de los descriptores de **TSS** en la **GDT** para las tareas inicial e idle.

- `TSS_INICIAL 13`
- `TSS_IDLE 14`

7.2. b

Para este punto definimos `tss_inicializar` en `tss.h/c` que crea la tabla de `tss_idle`, con `EIP` en `0x16000`, con el mismo **CR3** que el kernel, la misma pila, y los descriptores de segmento de datos y codigo de nivel 0.

7.3. c

Definimos `completar_tabla_tss` en `tss.h/c` que toma una tabla **TSS**, un puntero a codigo de la tarea y un puntero a un lugar donde guardar la **CR3**, luego dependiendo de donde se encuentre el codigo de la tarea lea asigna la posicion de inicio del jugar 1 o 2 (Desde 0x10000 las dos primeras paginas son codigo del jugador 1 y las otras 2 del jugador 2), crea un mapa de memoria usando `mmu_inicializar_dir_pirata` y lo asigna como **CR3** de la tabla, setea los selectores de segmento, los flags y guarda el **CR3** en una parte asignada de la primera pagina del area libre.

7.4. d, e

Definimos la funcion `agregar_descriptor_tss` en `tss.h/c` que toma como parametro un indice de la **GDT** y un puntero a un puntero a una tabla tss y agrega en la **GDT** un descriptor de **TSS** en el indice. Luego modificamos `tss_inicializar` y usamos `agregar_descriptor_tss` para agregar el descriptor de `tss_inicial` y `tss_idle`.

7.5. f

Para saltar a la tarea idle vamos a modificar `kernel.asm`, primero limpiamos `EAX` ya que vamos a pasar el descriptor de **TSS** de `tss_inicial` a ese registro, movemos el indice del descriptor a `AX` (`0x68`) y usando `LTR` lo cargamos en el registro especial de tareas. Luego hacemos un `JMP FAR` a el indice del descriptor de tareas en la **GDT** de `tss_idle` (`0x70`).

7.6. g

Para poder satisfacer lo pedido por el enunciado fue necesario agregar 3 funciones a `game.c`, luego, en base al codigo del *syscall* se modifico el *handler* para que salte a la funcion correspondiente. Las funciones son:

- `game_syscall_pirata_mover`
- `game_syscall_pirata_cavar`
- `game_syscall_pirata_posicion`

La primer funcion se encarga de mapear las paginas y copiar el codigo del pirata del cual se origino el syscall, siempre y cuando se este moviendo en una direccion permitida. En el caso de que mapee, esta funcion mapea a todas las tareas del jugador que lanzo inicialmente al pirata.

La segunda se encarga de cavar en la posicion actual, y devuelve un *char* que retorna `1` para indicar si la tarea cavo un area donde no se encontraban tesoros, ya que a esta altura no nos interesa desalojar, esta retorna siempre `0`.

Por ultimo, la tercer funcion se encarga de pedirle al pirata actual su posicion `X` e `Y`, retornandola de la forma que pide el enunciado.

7.7. h

Para poder lanzar un pirata de forma manual, alcanzo con crear una nueva funcion que llene la primer entrada libre de la **TSS** del jugador 1, luego se copio el codigo de la misma a la direccion fisica `0x551000` y esta se mapeo a la direccion virtual `0x400000` y se procedio a mapear al grilla de `3x3` del jugador 2.

Una vez hecho esto, cargado la `tss_inicial` y habiendo saltado a esta tarea, alcanza con saltar a la entrada de la **GDT** que apunte a la entrada de la **TSS** que creamos.

8. Ejercicio 7

8.1. a

Para el *scheduler* tenemos una estructura de datos que contiene los siguientes parametros:

- Nro. de tarea ejecutandose
- Proximo jugador (un caracter, puede ser `A` o `B`
- Modo *debug* activado
- Flag de excepcion (para el modo debug)
- Excepcion atendida (para imprimir la excepcion una unica vez en el caso de estar en modo debug)

El resto de los datos para poder manejar el cambio de tareas se encuentran dentro de la estructura de datos de cada jugador, estos tienen la siguiente informacion:

- Arreglo de piratas del jugador
- Posicion del arreglo del proximo pirata a ejecutar
- Posiciones mapeadas
- Arreglo de botines, este tiene la posicion del mismo y un flag que reporta si fue enviado un minero hasta el tesoro
- Cantidad de botines descubiertos
- Puntos hasta el momento
- Datos para imprimir el reloj de cada pirata

Por ultimo, cada pirata tiene los siguiente datos:

- Flag para representar si se encuentra actualmente en el juego
- Flag para representar si es pirata y no un minero
- Coordenadas `X` e `Y` de la posicion actual

Con estos datos tengo suficiente informacion para poder alterner entre los diferentes jugadores y los piratas de los mismos. Los valores iniciales dependen del puerto de donde salgan, pero siempre el jugador va a tener mapeado su puerto e inicialmente no va a tener piratas. Ya que ningun jugador se encuentra activo, se puede colocar cualquier jugador como proximo jugador, el *scheduler* se encarga de saltar apropiadamente a la tarea idle en tal caso.

8.2. b

Esta funcion opera bajo el siguiente pseudo-codigo:

```
1     if(proxJugador == 'A') {
2         if(jugadorA.piratas == 0) {
3             if(jugadorB.piratas == 0) {
4                 return IDLE;
5             } else {
6                 return PROX_PIRATA_B;
7             }
8         } else {
9             return PROX_PIRATA_A;
10    } else {
11        if(jugadorB.piratas == 0) {
12            if(jugadorA.piratas == 0) {
13                return IDLE;
14            } else {
```

```
15                  return PROX_PIRATA_A;
16              }
17          } else {
18              return PROX_PIRATA_B;
19          }
20      }
```

La idea detras de esta funcion es unicamente saltar a la tarea idle si no hay piratas en juego. Si hay aunque sea uno, siempre se retornara un codigo de tarea que no sea idle.

8.3. c, e, g

El mecanismo de cambio de tareas es relativamente sencillo, el trabajo principal lo hace `sched_tick`, la cual devuelve la entrada de la `GDT` de la tarea a saltar. Esta funcion responde al siguiente pseudo-codigo:

```
1       if(huboExcepcion & modoDebug) {
2           imprimirEstadoCPU();
3           return IDLE;
4       } else {
5           actualizar_relojes(tareaActual);
6           nuevoId = proximaTarea();
7           actualizar_jugador(proxJugador);
8           mineros_pendientes(proxJugador);
9           retun entradaGdt(nuevoId);
10      }
```

Vamos a analizar la funcion por partes.

- Si estamos en modo debug, y durante el ultimo tick de reloj se produjo una excepcion, tenemos que imprimir el estado de la misma, para esto tenemos los flags `modoDebug` y `huboExcepcion`. Como la ejecucion de las tareas debe interrumpirse alcanza con retornar el indice de la **GDT** de la tarea idle. La unica forma de reanudar la ejecucion de las tareas es bajando el flag `huboExcepcion`, este se baja mediante la tecla `y` del teclado.
- Si la ejecucion de tareas opera de forma normal, procedemos a hacer el cambio de tareas. Primero actualizamos los relojes, cambiando el del pirata que acaba de terminar su ejecucion.
- Toma el id de la nueva tarea a ejecutar, respetando la logica presentada en el punto `b` de este mismo ejercicio
- Gran parte de la funcionalidad del *scheduler* es hacer correctamente los cambios en las estructuras de datos pertinentes. Hacer esto implica cambiar el `proxJugador` y para el jugador que acaba de ejecutar su tarea, tengo que cambiar su `proxPirata`.
- El criterio de cambio de jugador es simple, si el otro jugador no posee piratas en juego, no cambio de jugador.
- Para cambiar el pirata del jugador que termino su ejecucion, alcanza con recorrer el arreglo de piratas del mismo en forma c̈ircular", es decir, si nos encontramos en el medio del mismo una vez que lleguemos al final, alcanza con volver al inicio y seguir recorriendo hasta que lleguemos a la posicion desde donde empezamos a recorrerlo. A medida que avanzamos, revisamos hasta encontrar el primer pirata en juego y lo seteamos como `proxPirata`, en el caso de que no encontremos otro, el `proxPirata` a ejecutar no cambiaria.
- En el caso de que se hubiera encontrado un tesoro, es necesario liberar un minero para juntar el botin, la funcion `mineros_pendientes` se encarga de revisar si el jugador que acaba de ejecutar encontro algun tesoro. Esto lo hace revisando el arreglo de botines del jugador en busca de algun botin no atendido, en caso de encontrarlo y si llega a haber un slot libre en los piratas del jugador, libera al minero en el puerto del jugador y marca al tesoro como atendido, en el caso contrario espera hasta el proximo tick del jugador para repetir el proceso.
- Por ultimo, se retorna el indice de la **GDT** de la proxima tarea a ejecutar

Para cerrar con este punto, vamos a hablar de los handlers de reloj y de teclado:

- En el caso del reloj, el *handler* se encarga de llamar a `sched_tick`, luego compara el valor retornado por la funcion con el valor actual del Task Register, si estos no coinciden salta a la nueva tarea

- La interrupcion de teclado se encarga de lanzar piratas y de activar el modo debug
- Si se toca la tecla para lanzar un pirata de algun jugador, el *handler* se encarga de revisar si este tiene algun slot libre, en caso de haberlo procede a ocuparlo con el nuevo pirata y llena la entrada de la **TSS** del slot libre con la informacion del nuevo pirata, el codigo del pirata es mapeado en la direccion del puerto del jugador y ademas se aumenta la cantidad de piratas del jugador en uno
- Si se toca la tecla de debug, el *handler* procede a revisar en que estado se encontaraba el flag `modoDebug`, en caso de estar seteado, revisa si el flag `huboExcepcion` estaba activado, en tal caso lo baja mantiendo el modo debug activo lo cual efectivamente reanuda la ejecucion de las tareas, si no estaba activado procede a bajar el flag de `modoDebug`, efectivamente saliendo de dicho modo. En caso de que `modoDebug` no hubiese estado seteado, procederia a activarlo, entrando entonces a ejecutar en modo debug.

8.4. d

El *handler* del syscall opera acorde a lo que pide el enunciado, para poder realizar sus funciones este utiliza la tarea actual del *scheduler* para determinar el pirata que llamo al syscall, y con eso puede realizar las acciones pertinentes.

8.5. f

Cuando se produce una excepcion, el *handler* de la misma se encarga de almacenar el estado del procesador al momento de ocurrir la excepcion, esta informacion la almacena en una estructura de datos. Una vez que almacena todo se procede a activar el flag de `huboExcepcion` del *scheduler*, se desocupa el slot del pirata (esto se determina mediante la tarea actual del *scheduler*) y se salta a la tarea idle.