TP3 - System Programming - Segundo Cuatrimestre 2020

Jerónimo Dacunda Ratti - 710/18 - jero.d.r22@gmail.com
José Gutiérrez - 459/00 - jagj77@gmail.com
Schiavinato Mauro - 299/19 - maurolschiavinato@gmail.com

Ejercicio 1

a) En `gdt.c` agregamos los indices en la tabla de descriptores de la GDT de los segmentos que había que agregar:

```
#define GDT_IDX_CODE_KERNEL 0xA
#define GDT_IDX_CODE_USER   0xB
#define GDT_IDX_DATA_KERNEL 0xC
#define GDT_IDX_DATA_USER   0xD
```

El enunciado explica que se deben direccionar los primeros 201MB de memoria fisica , por lo tanto el limite se expresa en paginas de 4KB y sera 0xC900:

0xC9 * 0x400 / 0x4 = 0xC9 * 0x100 = 0xC900

Los 4 segmentos se definen de forma muy similar, solo cambiando su nivel y si son de codigo o datos. El código fuente contiene comentarios sobre la elección de cada campo.

b)

1) Primero habilitamos la Gate A20 con el código de la catedra
2) Cargamos la GDT con segmentos definidos en **a)**, GDT_DESC
3) Ponemos en 1 el bit menos significativo de CR0
4) Modificamos el .asm para que el NASM genere instrucciones de 32 bits (BITS32)
5) Creamos una etiqueta en a la parte de 32 bits y realizamos un salto far usando como selector `GDT_IDX_CODE_KERNEL << 3`, que llamamos `GDT_OFF_CODE_KERNEL`, esto corresponde RPL = 0 y GDT = 0, IDX= 0xA, el segmento donde esta el codigo del kernel.
6) Creamos el selector del segmento de datos `GDT_IDX_DATA_KERNEL <<3`, que llamamos `GDT_OFF_DATA_KERNEL` y lo asignamos a SS, DS, ES, FS.
7) Seteamos esp y ebp a 0x25000, apartir de este punto podemos llamar a funciones en C.

c) En gdt.c declaramos un nuevo segmento , GDT_IDX_VIDEO_KERNEL cuyo valor es 0xE, el cual

```
BASE    + LEN    = LIMIT
0xB8000 + 0x8000 = 0xC0000
```

Y como esto es menor a 1Mb podemos definirlo en bytes.

d) Creamos una rutina `init_pantalla` encontrada en `kernel.asm` que usa el segmento GS que corresponde al selector de segmento 0x70 => Idx = 0xE, RPL=0, TI=0, para limpiar la pantalla y dibujar las partes verdes.

Desde esta rutina llamamos a la funcion `screen_draw_box` escrita en C que usa el segmento DS con su selector 0x60 => Idx = 0xC, RPL = 0 y TI= 0 para dibujar la caja azul y roja.

Resultado de inspeccionar GDT, vemos que se accesio solo a level 0 y la definición de segmentos corresponde a lo solicitado:

```
info gdt 0xA 0xE
Global Descriptor Table (base=0x0000000000003020, limit=279):
GDT[0x0050]=Code segment, base=0x00000000, limit=0x0c900fff, Execute/Read,
Non-Conforming, Accessed, 32-bit
GDT[0x0058]=Code segment, base=0x00000000, limit=0x0c900fff, Execute/Read,
Non-Conforming, 32-bit
GDT[0x0060]=Data segment, base=0x00000000, limit=0x0c900fff, Read/Write, Accessed
GDT[0x0068]=Data segment, base=0x00000000, limit=0x0c900fff, Read/Write
GDT[0x0070]=Data segment, base=0x000b8000, limit=0x000c0000, Read/Write, Accessed
```

Ejercicio 2

Definimos `#define GDT_SEL_CODE_KERNEL ( 0xA << 3 )` el selector del segmento de codigo del kernel, en la GDT con privilegio 0, lo usamos en la macro.

Revisemos los atributos de un Interrupt Gate.

La macro debe definir una entrada IDT de esta forma

```
| PARTE ALTA DE LA DIRECCION DE LA RUTINA | P=1 | DPL = 00 | 0 1 D=1 1 0 | 0 0 0 | 0 0 0 0 0 |
| GDT_SEL_CODE_KERNEL                     | PARTE BAJA DE LA DIRECCION DE LA RUTINA          |
```

Los attributos serian:

```
| 1 0 0  0 |  1 1 1 0 | 0 0 0  0 | 0 0 0 0 |
|   0x8    |   0xE    |   0x0    |   0x0   |
```

Definimos

```
#define INTERRUPT_GATE_ATTR 0x8E00
```

y lo utilizamos en la macro en idt_init para declarar las excepciones 0 a 8, 10 a 14 y 16 a 21.

Definimos las interrupciones en isr.asm e isr.h

luego modificamos kernel.asm para llamar a `idt_init` y luego cargar `IDT_DESC` con **lidt**. Luego de esto podemos ejecutar y revisar la idt cargada.

```
info idt

Interrupt Descriptor Table (base=0x0000000000004160, limit=2039):
IDT[0x00]=32-Bit Interrupt Gate target=0x0050:0x00001946, DPL=0
IDT[0x01]=32-Bit Interrupt Gate target=0x0050:0x00001950, DPL=0
IDT[0x02]=32-Bit Interrupt Gate target=0x0050:0x0000195a, DPL=0
IDT[0x03]=32-Bit Interrupt Gate target=0x0050:0x00001964, DPL=0
IDT[0x04]=32-Bit Interrupt Gate target=0x0050:0x0000196e, DPL=0
IDT[0x05]=32-Bit Interrupt Gate target=0x0050:0x00001978, DPL=0
IDT[0x06]=32-Bit Interrupt Gate target=0x0050:0x00001982, DPL=0
IDT[0x07]=32-Bit Interrupt Gate target=0x0050:0x0000198c, DPL=0
IDT[0x08]=32-Bit Interrupt Gate target=0x0050:0x00001996, DPL=0
IDT[0x09]=32-Bit Interrupt Gate target=0x0050:0x000019a0, DPL=0
IDT[0x0a]=32-Bit Interrupt Gate target=0x0050:0x000019aa, DPL=0
IDT[0x0b]=32-Bit Interrupt Gate target=0x0050:0x000019b4, DPL=0
IDT[0x0c]=32-Bit Interrupt Gate target=0x0050:0x000019be, DPL=0
```

```
IDT[0x0d]=32-Bit Interrupt Gate target=0x0050:0x000019c8, DPL=0
IDT[0x0e]=32-Bit Interrupt Gate target=0x0050:0x000019d2, DPL=0
IDT[0x0f]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x10]=32-Bit Interrupt Gate target=0x0050:0x000019dc, DPL=0
IDT[0x11]=32-Bit Interrupt Gate target=0x0050:0x000019e6, DPL=0
IDT[0x12]=32-Bit Interrupt Gate target=0x0050:0x000019f0, DPL=0
IDT[0x13]=32-Bit Interrupt Gate target=0x0050:0x000019fa, DPL=0
IDT[0x14]=32-Bit Interrupt Gate target=0x0050:0x00001a04, DPL=0
IDT[0x15]=32-Bit Interrupt Gate target=0x0050:0x00001a0e, DPL=0
IDT[0x16]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x17]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x18]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x19]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x1a]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x1b]=??? descriptor hi=0x00000000, lo=0x00000000
IDT[0x1c]=??? descriptor hi=0x00000000, lo=0x00000000
....
```

Ejecicio 3

a) Agregamos en el archivo `idt.c` e `isr.asm` las rutinas que corresponden.

b) Se agrega la rutina, para la interrupcion del reloj 32 que atiende el pic y llama a nextClock.

c) Escribimos la rutina `void printScanCode(uint8_t code)`, donde (code&0xF0)== 0x80 marca la tecla levantada con codigo (code&0x7F) o sea los 7 bits menos significativos son el codigo y el mas significativo marca si se suelta o presiona la tecla.

d) Se crean las rutina en para modificar en `idt.c` y `isr.asm`.

Ejercicio 4

Para inicializar el directorio y las tablas de páginas del kernel primero inicializamos el directorio todo en **not present**. Luego marcamos la siguiente pagina libre como 0x25000. Luego realizamos identity mapping a los primeros 4MiB.

La función **mmu__next__free__kernel__page** nos devuelve la página libre y va a incrementar el puntero que apunta a todas las páginas libres del kernel.

Para probar la paginación:

- Agregamos un breakpoint en el código de la tarea rick
- Luego llamamos a la funcion así, es como se va a comportar un morty pero se nos ocurrió probar usando el código de rick

```
mov eax ,cr3
push 0x1D00000 ; dirección virtual donde va a ir a parar
push eax       ; cr3 del kernel
push 0x4       ; 4 Páginas de código
push 0x10000   ; Código de rick esta acá
push 0x1C00000 ; dirección física donde va a ir a parar

call mmu_init_task_dir
```

```
add esp, 5*4

jmp 0x1D00000 ; debería ejecutar la tarea rick y parar justo en el breakpoint
```

Ejercicio 5

a) En `mmu_init` simplemente inizializamos el puntero de las páginas libres del kernel.

c) Creamos `mmu_init_task_dir` con los siguientes parámetros:

- paddr_t phy_start: lugar físico donde va a estar la tarea.
- vaddr_t virt_start: lugar virtual donde va a estar la tarea.
- paddr_t code_start: lugar donde está el código de la tarea.
- size_t pages: cantidad de páginas de la tarea.
- uint32_t task_cr3: cr3 de la tarea. Si es 0 entonces se crea uno nuevo.
- uint32_t code_cr3: cr3 del kernel.
- uint32_t attr: atributos para el mapeo de memoria.

Devuelve el cr3 creado si se le pasó 0 o en otro caso se devuelve el mismo que se le pasó por parámetro.

Ejercicio 6

a) Definimos:

```
#define GDT_SEL_TASK_INICIAL 0xF
#define GDT_SEL_TASK_IDLE    0x10
```

Vamos a usar los índices en el GDT para las tarea inicial y la tarea idle.

b/c/d/f) En tss_init inicializamos la tarea a los mismos valores, con el cr3 del kernel utilizando la funcion `make_tss` con los siguientes parámetros:

- phy_start: memoria física donde vamos a poner el código de la tarea.
- virt_start: memoria virtual que va a hacer referencia a la física de la tarea.
- code_start: memoria donde está el código de la tarea.
- pages: cantidad de páginas de la tarea.
- cr3_task: puntero a una variable que contiene el valor de cr3 de la tarea. Si es 0 el valor de la variable entonces creamos un cr3 y lo guardamos ahí.
- cr3_code: cr3 del kernel.
- is_user_task: 1 si es una tarea de usuario y 0 si es del kernel
- gdt_index_param: indice del descriptor a usar. Si es 0 entonces busca un descriptor libre en la gdt.

Lo que devuelve es el índice del descriptor que se usó. Utilizamos la función `make_tss` para crear todas las tareas, la de idle, la inicial, la de rick/morty y hasta la de los meeseeks. Esto es bueno porque lo hace más genérico al código, podemos crear tareas tanto de kernel como de usuario con la misma función y podemos decirle que nos cree un directorio de páginas o que utilice uno que le pasemos.

Esta función utiliza el descriptor de `gdt_index_param` a menos que sea 0, en ese caso busca un lugar disponible en la gdt para crear el descriptor de la tss de la tarea. Es interesante notar que todo el sistema está hecho para que si se quisiera agregar más meeseeks de los 10 que nos pide el enunciado, es simplemente cambiar el valor del define `AMOUNT_MSSEEKS_PER_PLAYER`. Notar igual que por enunciado la memoria virtual que nos dejan para mapear los meeseeks solo entran 10, asi que tendrían que dejarnos usar más lugar también ahí.

Ejercicio 7

a) En la gdt agregamos los descriptores de las tss de la tarea inicial y la tarea idle, así ya están fijas y podemos saltar desde `kernel.asm`. Luego en `tss_init` simplemente generamos las tablas tss y las asignamos a los descriptores de la tarea idle e inicial. Después en el inicio del scheduler lo único que hacemos es inicializar las estructuras donde nos guardamos las tareas de rick y morty y creamos ambas tareas.

b) Existen dos arreglos de estructuras, uno de tareas de rick y otro de tareas de morty.

Vamos a tener dos variables para saber que tarea estamos ejecutando en el scheduler:

- **currentMeeseekTask**: nos dice que número de meeseek de rick o morty se está ejecutando. Va a tener un valor de NOT_IN_MEESEEK_TASK si no está ejecutando un meeseek (es decir que está ejecutando un personaje principal, rick o morty).
- **currentTaskIsMortyOrMortyMeeseekTask**: nos dice si está en 1 que se está ejecutando morty o un meeseek de morty. En otro caso nos dice que se está ejecutando rick o un meeseek de rick.

En `sched_next_task` lo que hacemos es: Seleccionada la lista buscamos las primera tarea no visitada y viva, actualizamos sus contadores y retornamos su selector de no quedar vivas o a visitar, reseteamos las visitas, actualizamos el contador de la tarea rick/morty y su selector

e) Creamos la funcion kill_current_task que si es una tarea de un jugador simplemente termina el juego y si es una tarea de meeseek entonces matamos al meeseek, lo que significa que removemos el descriptor de la tss de la gdt, y eliminamos el meeseek de la estructura del schedular que guardaba a los meeseeks.

Ejercicio 8

g) Creamos la función `game_init` que realiza: - Crea las tareas rick y morty. - Inicializa las posiciones de las semillas. Vamos a tener un arreglo de estructuras que van a tener la posición X e Y de cada semilla y si ya fue tomada o no.

h) Para no tener que programarlas en assembler, todas las rutinas llaman a sus versiones de C que están definidas en `game.c`. Para `create_meeseek_c` realizamos los chequeos necesarios que nos piden en el tp: - Verificamos que no hay mas de AMOUNT_MSSEEKS_PER_PLAYER tareas por jugador - Verificamos que no haya ningún meeseek en esa posición, sino lo matamos al que está ahí (desmapeamos, removemos la tarea de la gdt y la removemos del scheduler). - No haya una megaseed, sino la asimilamos y no creamos el meeseek. - Verificamos que la dirección del código del meeseek que nos pasaron esté dentro de las direcciones virtuales de rick/morty. Luego de estas verificaciones creamos la tarea del meeseek simplemente con `make_user_task` que llama a `make_tss`. Para la memoria virtual utilizamos una fija para cada número de meeseek. Entonces el meeseek número 1 tomaría las primeras dos páginas a partir de `0x08000000`, el meeseek 2 tomaría las segundas dos páginas y así siguiendo. Siempre que se crea un meeseek se busca un lugar libre en el arreglo de meeseeks definido en `game.c` en la variable RickTasks o MortyTasks dependiendo del jugador.

Para la paginación:

l/i) `move_c` en game.c contiene la lógica de este punto. Para mover la tarea tenemos que primero mapear en modo kernel la posición física vieja y la posición física nueva ambas con identity mapping así las podemos copiar. Luego las compiamos con `__builtin_memcpy`. Por último desmapeamos el viejo lugar y mapeamos la nueva virtual con la nueva física con el cr3 de usuario de rick/morty dependiendo si el meeseek es de rick/morty. Para la restricción del movimiento simplemente tenemos un contador de cuantos ticks vivió el meeseek, el cual se aumenta en la función del scheduler. Este contador para no

superar ningún máximo hacemos que tengá un valor máximo. Mediante este contador podemos saber cuantos pasos puede moverse el meeseek.

j) `look_c` en game.c contiene la lógica de este punto. Simplemente hacemos búsqueda lineal para encontrar la megasemilla más cercana.

k) `use_portal_gun_c` en game.c contiene la lógica de este punto. Simplemente llama a la función de mover pero con parámetros de un meeseek contrario random.

m) `check_end_game` en game.c llamada en cada tick antes que otra acción. Si llego a terminar el juego entonces nos quedamos en un loop mostrando la pantalla con 'JUEGO TERMINADO'.

Eliminación de una tarea

Cuando ocurre alguna excepción en una tarea, esta debe morir. En nuestro código activamos una flag para que mate a la tarea actual, luego saltamos a la tarea idle y luego cuando ocurre la interrupción de reloj entonces matamos a la tarea.

Modo debugger

Primero que nada creamos una macro para las interrupciones que tienen código de error y para las que no. A partir de acá macros vemos que ambas pushean los valores que vienen pusheados de nuevo. Esto es porque vamos a llamar a una función genérica para ambas interrupciones, entonces en la que no tiene error code pusheamos un error code 0. A partir de acá ambas llaman `printDebug` que está definida al final del `isr.asm`. Acá simplemente lo que hacemos es pushear todos los registros y todos los valores que nos pedían que mostraramos en el debug. Tuvimos que hacer unas cuentas para el stack ya que necesitabamos el esp que nos viene pusheado antes. Notar que pedimos tres valores del stack, entonces simplemente vamos yendo para abajo en el stack (hacia posición mayor de memoria) y vamos obteniendo el stack hasta que lleguemos a tres valores o que el esp sea igual al ebp. El programa se tiene que compilar con `-fno-omit-frame-pointer`, de no ser el caso entonces no se va a pushear el ebp entonces va a tirar una excepción el debugger. Para el backtrace simplemente mostramos el ebp y vamos yendo para atrás en las llamadas hasta que ebp sea 0 y a partir de ahí mostramos todos ceros.

Buffer de video

Por como imprimimos en la pantalla, la cual reeimprimimos en cada tick de reloj imprimimos todo (fondo de nuevo, meeseeks luego y por último megasemillas), sucedía que los meeseeks y las semillas titilaban. Esto sucedía porque algunas veces la pantalla se refrescaba justo después que se imprimiera el fondo y antes de imprimir los meeseeks/megaseeds, entonces algunos frames no aparece y algunos si, haciendolo titilar. Por esto es que implementamos un buffer donde siempre imprimimos ahí y luego imprimimos todo el buffer junto en la pantalla, entonces nunca imprimimos el fondo si hay un meeseek arriba. Una alternativa hubiera sido imprimir al meeseek de nuevo cada vez que realiza un move, o se crea un meeseek. Como no había ninguna restricción de como realizar la muestra en pantalla entonces optamos por la anterior alternativa.

Opcional - Estrategia Rick/Morty

Como los meeseeks comparten la memoria virtual con morty, entonces podemos crear un único meeseek en la posición (0,0) que haga un look y avise al morty donde hay una megasemilla, asi luego el morty crea un meeseek ahí. Esto funciona porque el look da coordenadas relativas pero como está parado en el (0,0) entonces terminan siendo las absolutas. Como puede suceder que nos muevan de lugar,

haciendo inservible el look absoluto, entonces los meeseek que creamos simplemente van a hacer un look y morir. Y los meeseek que creamos para agarrar las semillas simplemente van a morir si no caen en una semilla. Dejamos el código de la estrategia en el archivo `estrategia.c`.