DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II
Segundo Cuatrimestre de 2016

Grupo: El Arquitecto

Integrante	LU	Correo electrónico
Freidin, Gregorio	433/15	gregoriofreidin@gmail.com
Taboh, Sebastián	185/13	sebi_282@hotmail.com
Romero, Lucía Inés	272/15	luciainesromero@hotmail.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. GDT

La Tabla de descriptores global o GDT es la encargada de organizar el sistema de segmentación entre otras cosas. Almacena descriptores los cuales pueden ser de segmento, de TSS, de LDT o de call gate. En nuestro sistema contamos con una estructura, gdt_entry, la cual completamos con descriptores de segmentos y de tss.

1.1. Descriptores de Segmento

Cada descriptor de segmento nos habilita distintas secciones de la memoria. En el caso del tp, nuestro sistema usa segmentación flat, por ende cada descriptor se maneja con el mismo rango de memoria, pero lo que cambia son los atributos con los que se accede a ella.

- Descriptor Nulo (requerido por Intel).
- Descriptores Nulos desde la entrada 1 hasta la 17 (requeridos por el tp).
- Descriptor de Segmento de Código de nivel 0 (privilegios de supervisor y lectura habilitada).
- Descriptor de Segmento de Código de nivel 3 (privilegios de usuario y lectura habilitada).
- Descriptor de Segmento de Datos de nivel 0 (privilegios de supervisor y escritura habilitada).
- Descriptor de Segmento de Datos de nivel 3 (privilegios de usuario y escritura habilitada).
- Descriptor de Segmento de video

Los descriptores previos tenían como base 0x0000, y como límite, 0x6fff, además, el bit de granularidad seteado, por ende cada uno de los segmentos referencia los primeros 1.75 GB. Excepto por el descriptor de segmento de video, el cual posee como base: 0xb800, límite: 0x0f9f y la granularidad en 0. Todos tenían el bit de sistema en 1 y el de presente también seteado.

1.2. Descriptores de TSS

Luego tenemos las entradas de las tareas, en las cuales profundizaremos más adelante.

- Descriptor de la Tarea Inicial.
- Descriptor de la Tarea Idle.
- De la entrada 25 hasta la 32, nos encontramos con descriptores de tss para cada una de las tareas de nuestro sistema.
- De la entrada 33 hasta la 40, nos encontramos con descriptores de tss para cada una de las banderas de nuestro tp.

Todos estos descriptores mantienen el siguiente formato:

Limite: 0x67

Base: Dirección de la tss

Presente: Seteado

Tipo: 0x9. Combinado con el bit de sistema en 0, tipo 9 se refiere a un descriptor de TSS.

Sistema: 0x0

DPL: 0x0. Esto es ya que en nuestro sistema queremos que las tareas sólo puedan ser accedidas por el kernel, es decir, que no se pueda "saltar" de una tarea a otra.

Granularidad: 0x0

AVL: 0x0

DB: 0x1 (32 bits)

L: 0x0

2. Modo Protegido

Al inicio de nuestro sistema nos encontramos en modo real. Esto es consecuencia de la condición de COMPATIBILIDAD de intel. Al iniciar nuestro sistema tenemos un 8086, nuestro código es de **16 BITS**; **NO** hay protección de memoria; podemos utilizar **TODAS** las instrucciones; AX,CX,DX **NO** son de propósito general. Y además, sólo podemos direccionar 1 MB de memoria. Por esto, luego de cargar los descriptores de código y datos vamos a pasar a "modo protegido", en el cual contamos con código de 32 bits, protección a memoria, y 4 GB de memoria direccionable.

Para poder realizar esto tenemos que encargarnos de un par de puntos previamente:

- Cargar el GDTR con la dirección de la GDT (LGDTR)
- Deshabilitamos las interrupciones **EXTERNAS** (CLI)
- Habilitamos A20, es decir, habilitamos el acceso a direcciones de memoria superiores a 2^{20}.
- Seteamos el bit PE del registro CR0. (PE: Protected Mode Enable).

Entonces, con el contexto ya armado, ejecutamos la instrucción: **JMP FAR [selector]:[offset]**. En el caso de nuestro TP, el selector sería 18<<3, ya que este es el descriptor de código de nivel 0; y como offset utilizamos una etiqueta la cual se encontraba inmediatamente a continuación de esta instrucción.

3. Paginación

El sistema de Paginacion de memoria, es una sistema en el cual se organiza la memoria de a paginas con tamaño 4K. Este sistema describe una funcion de mapeo de las direcciones virtuales, obtenidas a travez del sistema de Segmentacion, a direcciones fisicas de memoria. Agregando de paso, nuevos niveles de privilegio, como Supervisor o User, o en que direcciones se puede leer y escribir.

En este sistema optamos por implementar un sistema de paginacion en dos niveles. Esto significa que, para implementar el mapeo de memoria virtual a fisica se utilizan los siguientes elementos:

- **CR3:**

  En el registro de Control CR3, se va a tener guardado la direccion del Directorio de Paginas que esta actuando actualmente, en un sistema se pueden tener mucho mapeos diferentes, por lo que simplemente para cambiar el esquema de Paginación solo se tiene que remplazar el valor del CR3 por la direccion del Directorio de Paginas deseado.

- **Directorio de Páginas:**

  Esta estructura consiste en un arreglo de 1024 entradas, en las cuales cada PD_entry (entrada del Directorio de paginas) los primeros 20 bits son el prefijo de la direccion en la cual esta ubicada la Tabla de paginas correspondiente a dicha entrada, como estan alineadas a 4K cada direccion, los ultimos 12 bits de esta direccion se asumen que son 0. Y dentro de cada PD_entry se utilizan estos bits para establecer atributos, el bit 0 para marcar Presente, el bit 1 para marcar R/W, y el bit 2 para marcar U/S, como los principales atributos mencionados.

- **Tabla de Páginas:**

  Estra estructura consiste tambien en un arreglo de 1024 entradas, en la cual cada PT_entry los primeros 20 bits marcan la posicion donde comienza una pagina de 4K de direcciones fisicas de memoria. Nuevamente como las paginas son de 4K los últimos 12 bits de esta dirección se asumen que son 0, y dentro de la cada entrada de la PT, se utilizan para marcar atributos, los principales son los mismos que los mencionados para la PD

Proceso de Traducion Memoria Virtual ->Memoria Fisica

Sea D la dirección de la cual quiero obtener la direccion fisica mapeada, el proceso de obtencion es el siguiente.

Los primero 10 bits de D (D[31:22]), van a ser utilizados como indice dentro de la Page Directory, en la cual, de la entrada obtenida se va a sacar la direccion de la Page Table correspondiente a esta direccion.

Los segundo 10 bits de D (D[21:12]), van a ser utilizados como indice dentro de la Page Table, en la cual de la entrada obtenida se va a sacar la direccion de la pagina de 4K en donde esta contenida la direccion fisica mapeada a D

Por último, los ultimos 12 bits de D (D[11:0]), van a ser utilizados como Offset dentro de la pagina obtenida atravez de la Page Table. Osea la direccion final seria igual a direccion de pagina fisica 4K obtenida de PT + D[11:0]. Todo esto suponiendo que D paso todos los controles implementados en el sistema de paginación.

Paginación De Nuestro Trabajo Práctico

En primera instancia, inicializamos el mapeo para el codigo en kernel sobre el directorio de paginas ubicado en la direccion 0x27000, donde hacemos identity mapping desde la dirección 0x00000000 a 0x0077FFFF. Asignando en la seccion de atributos de cada entrada el valor de 0x3 que significa Presente = 1, R/W = 1, y nivel Supervisor. Las demas entradas las seteamos en 0.

Luego el esquema de paginación va a cambair según la tarea que se este ejectuando, para ello inicializamos varios directorios de paginas, en el cual generamos un mapping segun corresponda a la tarea que corra actualmente, donde mantenemos el mapeo hecho para el directorio del Kernel, pero agregando paginas nivel Usuario donde la tarea se va a ejecutar.

4. IDT

La IDT, Interruption Descriptor Table, se encarga de almacenar descriptores de rutinas de atención sobre interrpuciones de software, de hardware, y expeciones. Este se representa como un arreglo de maximo 2**13 entradas, de tamaño 64 bits. Estos Descriptores cargan informacion sobre, donde se hubica dicha rutina de atencion, atributos de presente, o DPL (nivel de privilegio requerido para acceder a dicha interrupcion), y el selector de segmento que se va a utilizar, en general este selector es el de Código nivel 0, ya que una interrpucion suele necesitar de accesos a nivel Kernel.

4.1. Rellenado de IDT

- **Expeciones:**

  Por restricciones de Intel, desde la posicion 0 hasta la 19 es donde se van a hubicar los descriptores de rutinas de atencion sobre expeciones, segun el orden que indica el manual de Intel. Estas tienen los atributos de presente en 1 y el DPL en 3, pues no queremos que ninguna tarea de nivel usuario llame a una Exepcion intencionalmente. Luego las entradas entre la 20 y la 31 estan reservadas a futuras posible exepciones, o usos que Intel les proporcione.

- **Interrupciones de Hardware:**

  Estas se ubican en las primeras entradas libres a partir de la 32, en particular la 32 es la interrupcion de Reloj, y la 33 la interrupcion de teclado. Ambas entradas en nuestro TP estan seteadas con el SegSel = Cod_L0, el bit de presente en 1, y DPL = 0.

- **Interrpuciones de Software:**

  Estas son seteadas a partir de posiciones mayores, en nuestro caso implementamos 2, en la entrada 80 y en la entrada 102. Donde la unica diferencia con las de Hardware respecto a los atributos, es que estas contienen DPL = 3, ya que queremos que puedan ser llamadas por tareas nivel usuario para ejecutar un codigo en nivel 0, esto en particular se le da el nombre de syscall.

4.2. Proceso de identificación de interrupción requerida

Cuando ocurre una interrpcion / exepción, el proceso basicamente es obtener el descriptor de interrupcion señalado por el Vector de Interrpciones, y de ahi saltar directamente a la rutina de atención. Ahora la forma en la que se carga dicho Vector es, para expeciones, el mismo procesador se encarga de indentificar cual es la posicion asignada a dicha interrupcion, dentro de la IDT segun Intel, y carga el vector con dicho valor. Para interrupciones de Hardware se utiliza lo que es el PIC de interrupciones donde cuando un elemento externo solicita una interrpucion, estos estan ordenados en un orden especifico segun un handler de hardware, y este identifica cual de los aparatos fue el que la solicito, cargando de esta manera al Vector de Interrupciones. Y por último a una interrpucion de Software simplemente es el numero q le sigue a la instruccion ïnt"que es la utilizada para solicitar una interrpucion especifica.

4.3. Excepciones

La rutina de atención a las excepciones es común a todos ellas. Esta se encarga de deshabilitar la tarea que ha caído en la interrupción, almacenar la información de la pila, y saltar a la tarea idle.

Para esto simplemente llamamos a la función **deshabilitar_tarea** con el índice del navío o la bandera que haya producido el error.

4.4. Interrupción de reloj

La rutina de atención a la interrupción de reloj llama a otras dos funciones; proximo_reloj y sched. La primera se encarga de la animación del relojito de cada tarea. Sched se encarga de saltar a la siguiente tarea, ya sea un navío o una bandera.

Cuenta con un contador el cual determina cuándo es momento de ejecutar todas las banderas y las funciones sched_proximo_indice y sched_proxima_bandera las cuales son explicadas es la sección **??**.

De esta manera sched pide el siguiente indice (de navío o bandera según corresponda) y salta a la tarea a la que este refiere. También contamos con un contador de tareas habilitadas, cuando este llega a 0 sched simplemente se limita a saltar a la tarea idle.

4.5. Interrupción de teclado

La interrupción 33 es la que se encarga del teclado, en nuestro caso sólo nos interesan 2 teclas de éste, **M** y **E**. La M nos muestra el mapa del juego, y la E el estado de los navíos y las banderas flameando.

A continuación presentamos un pseudocódigo de la rutina de atención a la int 33.

```
global _isr33
_isr33:
    pushad
    xor eax, eax
    in al, 0x60
    push eax
    call print_numerito
    add esp, 4
    call fin_intr_pic1
    popad
    iret
```

La instrucción **in al, 0x60** guarda en al el scan code almacenado en el puerto 0x60 y luego utilizamos esta información para escribir en pantalla con la función **print_numerito**.

Esta función se encargará de llamar a las funciones auxiliares necesarias para cumplir con lo establecido al principio de esta sección, mostrar el mapa si se presionó la tecla M, y la información de estados si fue la E.

Las funcione encargada de imprimir en pantalla el mapa utiliza la información almacenada en posicionTareas, el cual es una matriz de 8x3, la cual guarda la posición de la pantalla en la que se encuentra cada página de cada tarea, esto se actualiza en la función actualizar_mapa y también el valor de últimoMisil. La función encargada de imprimir la pantalla de estados utiliza la matriz paginasTareas, la matriz flags, el valor de ultimoError y la matriz de debug entre otras cosas. La primera, lamentamos si confunde su nombre, es una matriz que contiene las posiciones de cada pagina de cada tarea, pero estas en lugar de estar almacenadas como enteros, son una cadena de caracteres, de esta forma nos es más sencillo imprimirlos en la pantalla de estados. La información de la matriz debug nos la provee la pila cuando se produce algún error o alguna interrupción externa. Y en ultimoError se nos informa cuál fue el último error cometido, como bien nos dice su nombre.

4.6. Int 80 (0x50)

Esta es una syscall que sólo pueden utilizar los navíos, es conciderado un error si una bandera la llama. Esta syscall cuenta con 3 funcionalidades; **navegar**, **fondear** y **cañonear**. Depende de los parámetros con los que es llamada se ejecutan las distintas funcionalidades.

A continuación presentamos un pseudocódigo de la rutina de atención a la interrupción, lo presentamos en lenguaje C para hacer más sencillo su entendimiento.

```
if EAX == 0xAEF then
    navegar(current, cr3, ebx, ecx)
end if
if EAX == 0x923 then
    fondear(current, ebx, cr3)
end if
if EAX == 0x83A then
    cañonear(current, ebx, ecx)
end if
```

4.6.1. Navegar

Como su nombre lo indica, consiste en moverse a través del espacio marítimo, el cual en nuestro caso consiste en el espacio de memoria entre la dirección 0x100000 y la dirección 0x77FFFF. Los parámetros son 2 direcciones **físicas** las cuales serán mapeadas a las 2 páginas del navío respectivamente.

Para llamar a esta funcionalidad se debe escribir en EAX, 0xAEF. En EBX y ECX se pasan las direcciones físicas antes mencionadas.

A continuación presentamos un pseudocógido de la función navegar.

Algorithm 1 void navegar(uint current, uint cr3, uint ebx, uint ecx)

```
destinoPag1 = (unsigned char*) ebx
destinoPag2 = (unsigned char*) ecx
pagina1TareaActual = (unsigned char*) 0x40000000
pagina2TareaActual = (unsigned char*) 0x40001000
for i ← 0 .. 4095 do
    destinoPag1[i] ← pagina1TareaActual
    destinoPag2[i] ← pagina2TareaActual
end for
mmu_mapear_pagina(0x40000000, cr3, ebx)
mmu_mapear_pagina(0x40001000, cr3,ecx)
actualizar_mapa(ebx,ecx,1,current)
```

4.6.2. Fondear

Esta funcionalidad consiste en mover el ancla de lugar, y en nuestro TP consiste puntualmente en mapear la página asociada al kernel de nuestra tarea a otra dirección física entre el rango de 0x0x000000 hasta 0x0FFFFF.

Como ya vimos para llamar esta función EAX debe ser 0x923, y en EBX debemos pasar la nueva dirección **física** a mapear.

A continuación presentamos un pseudocógido de la función fondear.

Algorithm 2 void fondear(uint current, uint ebx, uint cr3)

```
mmu_mapear_pagina(0x40002000,ebx,cr3)
actualizar_mapa(ebx,0,0,current)
```

4.6.3. Cañonear

Cada uno de nuestros navíos cuenta con un cañon con el cual lanzar misiles. A esta funcionalidad se accede con EAX==0x83A y consiste en escribir 97 bytes desde la dirección pasada por EBX.

A continuación presentamos un pseudocógido de la función fondear.

Algorithm 3 void canonear(uint current, uint ebx, uint ecx)

```
misil = (unsigned char*) ecx
destino = (unsigned char*) ebx
for ( doi ← 0 .. 96 )
    destino[i] ← misil[i]
end for
actualizar_mapa(ebx,0,2,current);
```

4.7. Int 102 (0x66)

Esta syscall se encarga de salir de las banderas y saltar a la tarea idle. También se fija que ha sido llamada por una bandera y no por un navío, ya que si fue así, a este habrá que eliminarlo del scheduler junto con su bandera. Para esto llamamos a la función **inhabilitar_tarea** con el índice actual y también almacenamos la información de la pila para poder mostrarlo por pantalla en la sección de estados.

5. TSS

El Task State Segment o TSS, es una estructura especial del sistema utilizada para guardar los contextos de ejecucion de una tarea al momento de dejarla para saltar a otra. Y esta justamente comenzara con el contexto guardado en la TSS respectiva.

El tamaño de la TSS es de 104 Bytes, donde se guardan todos los registros de 32 bits, todos los selectores de segemento y el puntero a comienzo de la pila Nivel 0, 1, 2 con sus respectivos SegSel, que seran utilizados en caso de llamar a una interrupcion de diferente nivel a la tarea.

Proceso de Inicialización de Tareas

En primer lugar el sistema de multi-tasking requiere que haya una TSS destinada para la tarea inicial, esta TSS justamente es para cuando saltemos a la primer tarea a ejecutar el contexto actual, en el que estaba ejecutando el codigo, debe ser guardado en algun lado.

Luego como mencionamos anteriormente, cada tarea comienza a ejecutar segun lo que el contexto de su TSS asignada diga, por lo que no puede haber valores *fruta* para ciertos registros escenciales como CR3, EIP, CS, etc... Por ello dentro de la seccion de codigo de kernel hacemos llamado a la funcion **tss_inicializar** en la cual recorremos todas las TSS rellenando con los valores correspondientes a los elementos escenciales de la siguiente manera:

- **TSS_Tareas**

```
EIP = 0x40000000  - por enunciado
ESP = 0X40001C00  - por enunciado
EBP = 0X40001C00  - por enunciado
EFLAGS = 0X202  - permite interrupciones

DS = DAT_L3  - es tarea de nivel 3
CS = COD_L3  - es tarea de nivel 3

CR3 = PD correspondiente
```

- **Tss_Banderas**

```
EIP = Posicion correspondiente dentro de la funcion de la Tarea a la que corresponde
ESP = 0X40001FFC  - por enunciado
EBP = 0X40001FFC  - por enunciado
EFLAGS = 0X202  - permite interrpuciones

DS = DAT_L3  - es tarea de nivel 3
CS = COD_L3  - es tarea de nivel 3

CR3 = PD de la tarrea a la cual corresponde
```

Por último a cada Page Directory de cada tarea, los incializamos segun como pide el enunciado, haciendo identity mapping como en el kernel y agregando el mapeo a partir de la posicion 0x40000000, a las paginas fisicas donde se encuentra el codigo de cada tarea.

6. Scheduler

El scheduler es quien se encarga de determinar qué tarea debe ser ejecutada. Para esto contamos con dos arreglos de enteros de 8 posiciones, en el caso de nuestro TP, en el cual se almacena qué navíos/banderas pueden ejecutarse. Cada posición i representa al navío/bandera i+1, y si en dicha posición hay un 1 es que el navío/bandera continúa presente en el scheduler, y, si hay un 0, no. En el caso de nuestro TP algún navío o bandera puede cometer algún error, ya sea una excepción, en cuyo caso como vimos en la sección **??** el handler de la interrupción se encargará de ello; o, en el caso de las banderas, estas pueden haber sido interrumpidas por el reloj antes de pasar por la int 0x66. Cualquiera sea el caso, esto conlleva a la eliminación del navío y bandera correspondiente, por esto es que contamos con los arreglos que nos determinan las posibles "siguiente tarea".

Como bien vimos en la sección **??**, cuando debemos eliminar un navío y su bandera, lo único que hacemos es eliminar el índice correspondiente en el scheduler con la función **inhabilitar_tarea(uint error, int n)**, la cual además, nos imprime en la pantalla de estados el error que se ha cometido.

Como el scheduler se comprende de 2 arreglos, contamos con 2 índices, una para cada uno de ellos; **current** de navíos, y **currentBanderas**. Contamos con dos funciones las cuales nos proveen el siguiente navío o bandera a ejecutarse, **sched_proximo_indice** y **sched_proxima_bandera**. A continuación mostramos un pseudocódigo que aplica a ambas funciones pero con sus respectivos **currents** y con la diferencia que **sched_proxima_bandera** se reinicia al llegar a 8.

Algorithm 4 int sched_proximo_indice()

```
current += 1
i = 0
while tareas[current] %8 == 0 and i<9 do
    current += 1
    i += 1
end while
if tareas[current] == 0 then
    current = -1
end if
return current
```

Como vimos en la sección **??**, en nuestro TP la interrupción externa 32, la interrupción de reloj, funciona como regulador de los tiempos de las tareas. Cada navío tiene un tick de reloj para correr, si llama a una syscall, el tiempo restante de su quantum es destinado a la tarea idle. Las banderas, por su parte, también tienen un tick de reloj para ejecutarse, con la salvedad de que antes de cumplido este tiempo tienen que llamar a la syscall 0x66. En cada interrupción de reloj se determina cuál será la próxima tarea a ejecutarse.

En particular, en nuestro TP, tenemos determinado que cada 3 quantum de tareas se corren **TODAS** las banderas "disponibles", es decir, las que no han sido eliminadas del scheduler por algún tipo de error.

En resumen, las tareas/navíos se ejecutan en orden, cada 3 quantum se ejecutan las banderas que siguen en el scheduler, al cometer algún error el navío/bandera es eliminado del scheduler junto con su bandera/navío, la función proximo_indice nos devuelve el índice de la tarea a ejecutarse, si es que la hay.

¿Y qué sucede cuando no quedan tareas a ejecutarse?
Sigue corriendo indefinidamente la tarea idle. Situación que efectivamente ocurre en nuestro TP.