



**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	3
1.1. Descriptores de Segmento	3
1.2. Descriptores de TSS	3
2. Modo Protegido	4
3. Paginación	5
3.1. Proceso de Traducion Memoria Virtual ->Memoria Fisica	5
3.2. Paginación De Nuestro Trabajo Práctico	5
4. IDT	6
4.1. Rellenado de IDT	6
4.2. Proceso de identificación de interrupción requerida	6
4.3. Excepciones	6
4.4. Interrupción de reloj	6
4.5. Interrupción de teclado	7
4.6. Int 80 (0x50)	7
4.6.1. Navegar	7
4.6.2. Fondear	8
4.6.3. Cañonear	8
4.7. Int 102 (0x66)	8
5. TSS	9
5.1. Proceso de Inicialización de Tareas	9
6. Scheduler	10
7. Conclusión	11
7.1. Ancla	11
7.2. Ejercicio 4.8	11

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 tienen 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 los navíos 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:

Límite: `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)
- Deshabilitar las interrupciones **EXTERNAS** (CLI)
- Habilitar A20, es decir, habilitar el acceso a direcciones de memoria superiores a 2^{20} .
- Setear 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 es 18<<3, ya que este es el selector del segmento de código de nivel 0; y como offset utilizamos una etiqueta la cual se encuentra inmediatamente a continuación de esta instrucción.

3. Paginación

El sistema de Paginación de memoria organiza ésta de a páginas con tamaño 4K. Este sistema describe una función de mapeo de las direcciones virtuales o lineales, obtenidas a través del sistema de Segmentación, a direcciones físicas de memoria. Agregando de paso, nuevos niveles de privilegio, como Supervisor o Usuario, o en qué direcciones se puede leer y escribir.

En nuestro TP optamos por implementar un sistema de paginación en dos niveles. Esto significa que, para implementar el mapeo de memoria virtual a física se utilizan los siguientes elementos:

- **CR3:**

En el registro de Control CR3, se va a tener guardado la dirección del Directorio de Páginas que está actuando actualmente, en un sistema se pueden tener muchos mapeos diferentes, por lo que simplemente para cambiar el esquema de Paginación solo se tiene que reemplazar el valor del CR3 por la dirección del Directorio de Páginas deseado.

- **Directorio de Páginas:**

Esta estructura consiste en un arreglo de 1024 entradas, en las cuales cada PD_entry (entrada del Directorio de páginas) los primeros 20 bits son el prefijo de la dirección en la cual está ubicada la Tabla de páginas correspondiente a dicha entrada. Como cada página está alineada a 4K los últimos 12 bits de esta dirección 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, etc. Los mencionados son los que utilizamos en nuestro TP.

- **Tabla de Páginas:**

Esta estructura consiste también en un arreglo de 1024 entradas, de cada PT_entry los primeros 20 bits marcan la dirección donde comienza una página de 4K, y dicha dirección es física. Nuevamente como las páginas son de 4K los últimos 12 bits de esta dirección se asumen que son 0, y dentro de la cada entrada de la PT, estos se utilizan para marcar atributos, los principales son los mismos que los mencionados para la PD.

3.1. Proceso de Traducción Memoria Virtual -> Memoria Física

Sea D la dirección virtual de la cual quiero obtener la dirección física mapeada, el proceso de obtención es el siguiente.

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

Los segundos 10 bits de D (D[21:12]), van a ser utilizados como índice dentro de la Page Table, en la cual de la entrada obtenida se va a sacar la dirección de la página de 4K.

Por último, los últimos 12 bits de D (D[11:0]), van a ser utilizados como Offset dentro de la página obtenida a través de la Page Table. Osea la dirección final sería: dirección de página física obtenida de PT + D[11:0]. Todo esto suponiendo que D pasó todos los controles del sistema de paginación.

3.2. Paginación De Nuestro Trabajo Práctico

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

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

4. IDT

La IDT, Interrupt Descriptor Table, se encarga de almacenar descriptores de rutinas de atención sobre interrupciones de software, de hardware, y excepciones. Éste se representa como un arreglo de, como máximo, 2^{13} entradas, de tamaño 64 bits. Estos Descriptores contienen información sobre, por ejemplo; dónde se ubica dicha rutina de atención; atributos de presente; atributos de DPL (nivel de privilegio requerido para acceder a dicha interrupción); el selector de segmento que se va a utilizar, el cual, en general es de Código de nivel 0, ya que una interrupción suele necesitar de accesos a nivel Kernel.

4.1. Rellenado de IDT

- **Excepciones:**

Por restricciones de Intel, desde la posición 0 hasta la 19 es dónde se van a ubicar los descriptores de rutinas de atención sobre excepciones según el orden que indica el manual de Intel. Éstas tienen los atributos de presente en 1 y el DPL en 3, pues no queremos que ninguna tarea de nivel usuario llame a una excepción intencionalmente. Luego las entradas entre la 20 y la 31 están reservadas a futuras posibles excepciones, o usos que Intel les proporcione.

- **Interrupciones de Hardware:**

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

- **Interrupciones de Software:**

En nuestro caso implementamos 2, en la entrada 80 y en la entrada 102. Donde la única diferencia con las de Hardware respecto a los atributos, es que éstas contienen DPL = 3, ya que queremos que puedan ser llamadas por tareas nivel usuario para ejecutar un código en nivel 0, a esto en particular se le da el nombre de syscall.

4.2. Proceso de identificación de interrupción requerida

Cuando ocurre una interrupción el procesador obtiene el descriptor de ésta, obtiene el descriptor de segmento que se especifica en éste y así, salta a su rutina de atención añadiendo el offset que ofrece el primero.

En el caso de las excepciones, es el procesador el que se encarga de identificar cuál es la posición asignada a esa interrupción en la IDT; para interrupciones de Hardware es el PIC quien se encarga; y, por último, en cuanto a las interrupciones de Software, estas se invocan con la instrucción **INT 0x??**, por ende el inmediato que acompaña la instrucción es el índice de la IDT.

4.3. Excepciones

La rutina de atención a las excepciones es común a todas ellas. Ésta 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 con las funciones **sched_proximo_indice** y **sched_proxima_bandera** las cuales son explicadas en la sección 6.

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

La función 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 `ultimoMisil`.

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 página de cada tarea, pero éstas, 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)

Ésta es una syscall que sólo pueden utilizar los navíos, es considerado un error si una bandera la llama. Ésta 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 virtuales 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 pseudocódigo 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  $i \leftarrow 0 \dots 4095$  do
    destinoPag1[i]  $\leftarrow$  pagina1TareaActual
    destinoPag2[i]  $\leftarrow$  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, lo que en nuestro TP se traduce puntualmente a mapear la página asociada al kernel de nuestra tarea a otra dirección física entre el rango de 0x0x000000 hasta 0x0FFFFFF.

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 pseudocódigo 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 hasta EBX+97.

A continuación presentamos un pseudocódigo de la función fondear.

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

```

misil = (unsigned char*) ecx
destino = (unsigned char*) ebx
for (  $doi \leftarrow 0 \dots 96$  )
    destino[i]  $\leftarrow$  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 ejecución de una tarea al momento de dejarla para saltar a otra. Y ésta justamente comenzará 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 segmento y el puntero al comienzo de la pila de Nivel 0, 1, 2 con sus respectivos SegSel, que serán utilizados en caso de llamar a una interrupción de diferente nivel a la tarea.

5.1. Proceso de Inicialización de Tareas

En primer lugar el sistema de multi-tasking requiere que haya una TSS destinada para la tarea inicial; el propósito de ésta TSS es el de guardar el contexto previo a saltar a la primera tarea a ejecutar.

Luego como mencionamos anteriormente, cada tarea comienza a ejecutar según el contexto de su TSS asignada, por lo que no puede haber valores *fruta* para ciertos registros esenciales como CR3, EIP, CS, etc... Por ello dentro de la sección de código de kernel llamamos a la función `tss_inicializar` en la cual recorreremos todas las TSS y las rellenamos con los valores correspondientes a los elementos esenciales 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 interrupciones

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 inicializamos según como pide el enunciado, haciendo identity mapping como en el kernel y agregando el mapeo a partir de la posición 0x40000000, a las páginas físicas a partir de la dirección 0x100000 y a éstas copiamos el código de las tareas el cual se encuentra a partir de la dirección física 0x10000.

6. Scheduler

El scheduler es quien se encarga de determinar qué tarea debe ser ejecutada. Para esto contamos, en el caso de nuestro TP, con dos arreglos de enteros de 8 posiciones en los cuales 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 4 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 4, 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, uno para cada arreglo; **current** de navíos, y **currentBanderas**. Contamos además 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 4, 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, los navíos se ejecutan en orden, y, 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 sino devuelve -1.

¿Y qué sucede cuando no quedan tareas a ejecutarse?

Sigue corriendo indefinidamente la tarea idle. Situación que efectivamente ocurre en nuestro TP. Al ocurrir esto el contador de tareas habilitadas está en 0, y la función **sched** del handler de la interrupción de reloj simplemente retorna a éste; ya que sería un error intentar saltar a la tarea actual. Misma situación se da cuando queda una sola tarea en ejecución.

7. Conclusión

7.1. Ancla

Queremos aportar a la cátedra nuestro siguiente descubrimiento:

Respondiendo al punto optativo **4.8 inciso d**, el ancla de la siguiente imagen, misma imagen que la figura 2 del enunciado del TP; se encuentra en la esquina de la intersección de las calles Pedernera y Tte. Cnel. Casimiro Recuero en el barrio porteño de Flores.



Figura 1: Ancla

7.2. Ejercicio 4.8

Luego de implementar lo descrito en este informe, sumado a ciertas funciones para imprimir en pantalla, creemos que nuestro sistema operativo cumple con lo establecido en el TP. Sin ir más lejos nos sentimos preparados y ansiosos por consagrarnos victoriosos en el evento de "Guerra de Tareas", ya que un navío letal está siendo diseñado para dicho propósito.