



**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. Ejercicio 1	3
2. Ejercicio 2	4
3. Ejercicio 3	5
4. Ejercicio 4	6
5. Ejercicio 5	7
6. Ejercicio 6	8
7. Ejercicio 7	9

1. Ejercicio 1

El primer ejercicio consistió en dos elementos, el primero fue sobre llenar la Global Descriptor Table (GDT), con ciertos segmentos y el segundo, en pasar a modo protegido.

Inicializar la GDT

En esta parte lo que hicimos primero fue crear un arreglo de `gdt_entry`, en el archivo `gdt.c` de 41 entradas y luego completamos 6 de las mismas. En la primera posición del arreglo fue seteado el descriptor Nulo por restricciones de Intel. Y los otros 5 descriptors de segmento fueron completados empezando desde la posición 18, por restricciones del TP, hasta la 22 inclusive.

Los dos descriptors de segmento de las posiciones 18 y 19 fueron seteados como segmentos de código nivel 0 y 3 respectivamente. Empezando la base de ambos desde la posición `0x0000`, y con un límite de `0x6ffff` con el bit de Granularity activado representando así 1.75 GB.

1.75 GB son $\frac{7}{4} * 2^{30}B = 7 * 2^{28}B$. Como hay 20 bits del campo "Límite" no iban a ser suficientes para direccionar 1.75 GB como se buscaba, entonces por eso se puso un 1 en el bit de Granularidad para que el Límite se refiriera a bloques de 4 KB. Así, se obtiene $\frac{7 * 2^{28}B}{4KB} = 7 * 2^{16}$ y restándole 1 se consigue `0x6ffff`, el contenido que se puso en el campo "Límite".

Al type de ambos segmentos se les puso el valor de `0xA` (Execute/Read), y los atributos de Sistema y Presencia en 1.

Los dos descriptors de segmento de las posiciones 20 y 21 fueron seteados como segmentos de datos nivel 0 y 3 respectivamente. Empezando la base de ambos desde la posición `0x0000`, y con un límite de `0x6ffff` con el bit de Granularity activado. Al type de ambos segmentos se les puso el valor de `0x02` (Read/Write), y los atributos de Sistema y Presencia en 1.

Por último el descriptor seteado en la posición 22 del arreglo, fue colocado como un descriptor de video, con la base a partir de `0xb8000`, y límite `0xf9f`, con el bit de Granularity en 0. A este segmento también se le asignó en los atributos de Sistema y Presencia el valor 1.

Pasar a Modo Protegido

Luego de completar la GDT, creándola con los requisitos ya especificados, y habiendo deshabilitado las interrupciones con la instrucción `cli`, se cargó el GDT register usando `lgdt`.

Después habilitamos la línea A20 para poder acceder a las posiciones mayores a 2^{20} y seteamos el bit de PE de CR0 en 1.

Así ya con todo el contexto armado, ejecutamos la instrucción `jmp 18 << 3:mp` para hacer un `jmp` far a modo protegido donde ponemos dentro del selector de segmento de código el valor 18 que es la posición dentro de nuestro arreglo `gdt`, donde está el descriptor de código nivel 0.

Segmento de pantalla

Agregamos una entrada a la GDT para describir el segmento de video.

El tamaño era $80 * 25 * 2$ por lo que al restarle 1 se obtuvo `0xf9f`, dato con el que se completó el campo "Límite", con "Granularidad" en 0. En "Base" se puso `0xb8000` y se asignó al bit "S" un 1 pues el tipo del descriptor era de datos y al campo "Tipo" `0x2` dado que era de sólo lectura y accedido.

Limpiar y pintar la pantalla

2. Ejercicio 2

Inicialización de la IDT

En este ejercicio el tema que se va a tratar es el de inicializar la IDT. Para ello, primero se declaró la estructura de la IDT en `idt.c`, como un arreglo de 256 entradas de `idt_entry`.

En este punto lo que hicimos fue simplemente llenar las primeras 31 entradas de este arreglo, con las interrupciones definidas por Intel. Para ello se utilizó una macro, ya definida, en la cual asignamos a todas las entradas por igual, como segmento de código en el cual se van a ejecutar, el número 18 dentro de la GDT, por ser este de nivel 0. Y en los atributos, el valor de `0x8E00`, lo cual especifica que cada excepción tiene el bit de presencia encendido, nivel de privilegio 0, y que es una interrupción de 32 bits. Y luego a cada entrada dentro del campo `offset`, le escribimos la dirección del handler de la misma.

Los handler de las interrupciones fueron declarados en `isr.h` e implementados en `isr.asm`. Donde a cada uno, simplemente se le pasó la tarea de imprimir por pantalla el número de interrupción que representa. Para ello en el área de memoria de la pantalla imprimimos el texto que según Intel representa dicha interrupción.

3. Ejercicio 3

Buffer de video

Inicialización del directorio y de la tabla de páginas para el *kernel*

Activación de paginación

4. Ejercicio 4

Inicialización del directorio y de la tabla de páginas para tareas

Mapeo y desmapeo de páginas de memoria

Construcción del mapa de memoria para tareas

5. Ejercicio 5

Entradas en la IDT

Por el posible caso de que se modificaran los registros dentro de la interrupción, será necesario salvar el estado de los registros al entrar en la rutina (con la instrucción **pushad**) y restaurarlos antes de salir de ella (con la instrucción **popad**).

Rutina asociada a la interrupción de reloj

Después de llamar a la rutina **proximo_reloj**, que se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla, y antes del fin de la rutina, debemos rehabilitar las interrupciones, y esto lo hacemos con la instrucción **call fin_intr_pic1**.

```
global _isr32
_isr32:
    pushad
    call proximo_reloj
    call fin_intr_pic1
    popad
    iret
```

Rutina asociada a la interrupción de teclado

Esta rutina primero lee utilizando la instrucción **in al, 0x60** el valor almacenado en el puerto 0x60, que dependerá de si la acción realizada en el teclado sea apretar una tecla o soltarla y de qué tecla sea.

Este valor queda en el registro **eax**, que se pushea para luego llamar a **print_numerito**, una función que realiza lo que corresponda sobre la pantalla.

Finalmente, se informa al PIC que se pueden rehabilitar las interrupciones.

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

Rutinas asociadas a la interrupciones 0x50 y 0x66

6. Ejercicio 6

A cada tarea le corresponden dos TSS, una para el código de la tarea y otra para la ejecución de la función “bandera”.

Definición de las entradas en la GDT para las tareas

Cada tarea creada necesita tener su descriptor de TSS en la GDT. Este nos permite acceder a la TSS de cada tarea, que definimos completando los campos del struct tss que se declara en tss.h. Guardamos la tarea `_inicial` en la posición 23 de la GDT y la tarea `Idle` en la posición 24 (valores referenciados respectivamente por las constantes globales `TAREA_INICIAL` y `TAREA_IDLE` definidas en `defines.h`). Las nuevas entradas de la GDT son inicializadas al llamar desde `kernel.asm` a la función `tss_inicializar` implementada en `tss.c`.

Se completaron los campos de los descriptores de las TSS de la siguiente forma:

- **Límite:** Como el tamaño de la estructura tss es 104, al restarle 1 obtenemos 0x67.
- **Tipo:** Este campo tiene 4 bits: 10B1, donde B (*busy*) representa que esté en uso la tarea asociada a este descriptor. Inicialmente la tarea no estará cargada así que B se seteará en 0, por lo que Tipo contendrá 0x9.
- **Nivel de privilegio:** Queremos que sólo el *kernel* pueda realizar el cambio de tarea a esta, por lo que completamos con 0x0.
- **Presente:** Este campo tiene 0x1 dado que hay una tss asociada a este descriptor.
- **System:** En este campo se pone 0 cuando se trata de contenidos de sistema como ser gdt, ldt, tss y 1 cuando son datos o código, por lo que se puso 0x0.
- **db:** Como se trabaja en 32 bits, se cargó 0x1.

Los demás campos se cargaron en 0.

TSS de Idle

Como la tarea `Idle` se encuentra en la dirección 0x00020000, pusimos en `eip` 0x20000. Además, como la pila se aloja en la página 0x0002A000 y será mapeada con identity mapping, completamos `ebp` y `esp` con 0x2A000. `EFLAGS` se completó con 0x202.

TSS de las demás tareas

Como el código de las tareas está mapeado a partir de la dirección 0x40000000 completamos el `eip` de los navíos con esa dirección.

Código para la ejecución de la tarea Idle

7. Ejercicio 7

Inicialización de las estructuras de datos del *scheduler*

Función `sched_proximo_indice()`

Primero declaramos dos arreglos de 8 posiciones, uno de tareas y otro de banderas, para indicar si las tareas estaban vivas. Además, usamos un contador `current` para saber qué tarea correspondía correr.

```
int  tareas[8]={1,1,1,1,1,1,1,1};
int  banderas[8]={1,1,1,1,1,1,1,1};
int  current = -1;
```

Luego implementamos la función `sched_proximo_indice()`. Esta suma 1 a `current` y entra en un ciclo en el que se aumenta 1 mientras se encuentren tareas que no estén vivas. Si se recorren todas y no hay ninguna para correr entonces `current` vuelve a -1, en otro caso queda en `current mod 8`.

```
short sched_proximo_indice() {
    current +=1 ;
    int i = 0;
    while(tareas[current%8] == 0 && i < 20){
        current +=1;
        i = i+1;
    }
    if(i == 20) return -1;
    return current %8;
}
```