



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming

Organización del Computador II
Primer Cuatrimestre de 2016

Grupo: Yo no manejo el raiting, yo manejo un Rolls-Royce

Integrante	LU	Correo electrónico
Luis Enrique Badell Porto	246/13	luisbadell@gmail.com
Nicolas Bukovits	546/14	nicko_buk@hotmail.com
Kevin Frachtenberg	247/14	kevinfra94@gmail.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>

Resumen

En el siguiente trabajo práctico, se realizó una implementación de un sistema operativo básico en el cual se aplicaron los conceptos de System Programming vistos en clase. El sistema desarrollado corre en Bochs (programa que permite simular una computadora IBM-PC y realizar tareas de debugging) y permite correr tareas de usuario concurrentemente. La implementación del sistema fue realizada siguiendo una serie de ejercicios en los cuales se desarrolló de forma gradual el sistema completo. La resolución de los ejercicios y las decisiones tomadas están explicadas en este informe.

Índice

1. Resolución de Ejercicios	3
1.1. Ejercicio 1	3
1.2. Ejercicio 2	4
1.3. Ejercicio 3	4
1.4. Interrupciones	5
1.5. El retorno de la GDT y la aparición de la TSS	5
1.6. El Scheduler	5
2. Conclusiones y trabajo futuro	5

1. Resolución de Ejercicios

1.1. Ejercicio 1

a) En esta primera parte se empezó a completar la GDT, que contiene los descriptores globales del sistema. El archivo modificado fue el `gtd.c` que contiene el código de implementación de la GDT. Por normas de Intel, la primera entrada en la GDT tiene que ser nula, por lo que la misma se completó con todos sus valores en 0. Posteriormente se agregaron 4 segmentos requeridos por el enunciado: un segmento para código de nivel 0 (kernel), en el cual se configuró el atributo `dpl` (privilegio) en 0 para que sólo pueda ser usado por el Kernel, otro segmento para datos también de nivel 0 para el Kernel, un segmento de código de nivel 3 (usuario con `dpl` 3) y otro segmento de datos de nivel 3. Por requerimiento del enunciado todos estos selectores de segmento fueron configurados para que direccionen los primeros 878 MB de memoria, por lo que los atributos de límite y base son iguales para todos. Las diferencias que presentan son los atributos de `dpl` antes mencionados y del tipo de selector que fue cargado con 8 para código y 2 para datos.

b) En este punto del ejercicio se implementó el cambio de modo real a protegido y se seteó la pila del kernel en la dirección `0x27000`. Este cambio fue realizado en el archivo `kernel.asm` que contiene el código `asm` que corresponde al kernel. El código del mismo comienza inhabilitando las interrupciones (mediante la instrucción `cli`) y cambiando el modo de video a `80 x 50`. El código que se agregó para cumplir con el requerimiento en este punto fue habilitar `A20`, para lo cual sólo fue necesaria llamar mediante a un `call` a una función externa que ya estaba implementada y se encarga de dicha tarea; y cargar el registro `GTDR` que contiene la dirección física en donde se encuentra la GDT. Para ello se utilizó la instrucción `lgdt` a la cual se le pasó como argumento la dirección de la `gdt`. Posteriormente fue necesario habilitar el modo protegido para lo cual era necesario que el bit de `PE` (que indica si el modo protegido está habilitado o no) del registro de control `CR0` sea seteado en 1. Como no está permitido operar directamente con el registro `CR0` se copió su valor al registro de propósito general `eax` mediante la instrucción `MOV` y posteriormente se realizó una operación `OR` del registro `eax` con el inmediato de destino en 1, para que el último bit sea seteado en 1. Finalmente se copió el valor actualizado de `eax` en `CR0` y se cambió el modo a protegido realizando un `jump far`. La instrucción utilizada fue `jmp 0x20:mp` donde `0x20` representa el selector de código de nivel 0. El cálculo realizado fue el siguiente: índice de código de nivel 0 (que es el 4) multiplicado por 8, es decir shiftado 3 lugares a la izquierda. El `mp` es una etiqueta definida en la próxima instrucción a ejecutar justo inmediatamente después del `jump`, en donde comienza el código que a partir de ese momento se va ejecutar en modo protegido. Las primeras instrucciones que se ejecutan en este punto se encargan de cargar los selectores de segmento que tienen que estar cargados porque es requerimiento del modo protegido. Se cargaron entonces los registros `ds`, `es`, `gs`, `fs` y `ss` con el índice de datos de nivel 0 de la GDT (segmentación `flat`). El `cs` (code segment) no fue necesario cargarlo debido a que la instrucción `jmp` lo cargó. Se cargaron además los registros `esp` y `ebp` de la pila con el valor `0x27000` que es la dirección que indicaba el enunciado en donde está la pila del Kernel (`MOV esp, 0x27000` y `MOV ebp, 0x27000`).

c) Se agregó un segmento adicional en la `gdt`, que describe el área de la pantalla en memoria que va a ser utilizada solamente por el kernel. El segmento que se agregó por lo tanto fue configurado con el `dpl` en 0 y es de tipo 2. La GDT entonces pasa a tener en este punto 6 segmentos (teniendo en cuenta al primer segmento nulo).

d) El enunciado solicita que se pinte el área del mapa en un fondo de color. Se implementó una macro en el archivo `imprimir.mac` denominada `pintarPantalla` la cual se encarga de pintar el fondo del mapa de color gris. El código de la misma es muy simple ya que se trata de un ciclo que copia a memoria en las posiciones del mapa, en los bytes de atributos de cada pixel, el valor correspondiente al color de fondo de gris. El código de la rutina es el siguiente:

```
%macro pintarPantalla 0
```

```
    push ecx
    push edi

    mov ecx , 4000
    xor edi , edi
%%cicloGris:
    mov word [edi*2 + 0x000B8000] , 0x7700 ;
    inc edi
    loop %%cicloGris
```

```
    pop edi
    pop ecx

%endmacro
```

1.2. Ejercicio 2

a) En esta parte de la implementación se tuvo que completar las entradas en la IDT (que contiene los descriptores de las interrupciones) para asociar diferentes rutinas a las excepciones del procesador. Los archivos modificados fueron el `idt.c` y el `isr.asm`. Se implementó la función `idt_inicializar()` en `idt.c` la cual se encarga de inicializar las excepciones. Las excepciones configuradas en este punto fueron las contenidas en el rango 0-19 y para configurarlas se definió una macro llamada `IDT_ENTRY(numero)` cuya implementación es la siguiente:

```
#define IDT_ENTRY(numero)
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);
    idt[numero].segsel = (unsigned short) GDT_OFF_IDX_DESC_CODE0;
    idt[numero].attr = (unsigned short) 0x8E00;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xFFFF);
```

La misma se encarga de definir en `idt`, que es un arreglo de 256 posiciones que contiene las descripciones de las excepciones, en la posición indicada por `numero`, el `offset`, los atributos y el selector de segmento que se trata de un selector de código de nivel 0 ya que van a ser rutinas que van a correr el kernel. Dentro de `idt_inicializar()` sólo se tuvo que llamar 20 veces a `IDT_ENTRY(i)` siendo `i` el número de excepción comprendido en el rango de 0-19. Por otro lado, el archivo `isr.asm` que contiene el código `asm` de las rutinas de atención de interrupciones fue modificado agregando una macro `_isr`: la cual es llamada cuando se produce una excepción y en la misma se usaba el número de excepción que se produjo para mostrar un mensaje en pantalla de la excepción. Para ello se definieron mensajes para cada una de las excepciones. Por ejemplo para el caso de la división por cero se agregó el siguiente mensaje:

```
msj0: db'Divide Error!'
msj0_len equ $ - msj0
```

b) Se solicita que se implemente lo necesario para que se pueda utilizar la IDT y que se pruebe generando una excepción. Para ello en el archivo `kernel.asm` se realiza un `CALL` a la función `idt_inicializar` implementada en el punto anterior y luego se carga el registro `IDTR` que contiene la dirección física en donde se encuentra alojada la IDT. La instrucción para cargar la misma es `lidt` a la cual se le pasa como argumento la dirección de la `idt`. Con estas modificaciones el sistema ya estaba configurado para manejar las excepciones. Para probarlo se agregaron tres instrucciones cuyo propósito era dividir por cero para que se produzca la excepción de `Divide Zero`. El código agregado fue:

```
xor ebx, ebx
xor eax, eax
div eax
```

El mismo produjo una excepción de dividir por cero, con lo que pudimos probar empíricamente que las excepciones estaban siendo manejadas y funcionaban correctamente. Dichas líneas de código fueron comentadas luego de que se corroboró que funcionaba para que no se produzca la excepción y el kernel siga ejecutando normalmente, y si eventualmente en algún momento se deseaba volver a verificar su funcionalidad sólo se tengan que descomentar. Además nos sirvió como documentación y registro de lo que se fue implementando.

1.3. Ejercicio 3

a) En este punto se pide implementar una rutina que limpie el buffer de video y lo pinte como indica el enunciado. Para esto, se realizó una macro definida en el archivo `imprimir.mac` cuyo propósito es pintar la pantalla como indica el enunciado. Dentro de esta macro se utiliza la otra macro previamente definida de `pintarPantalla`.

b) En este punto se tuvo que realizar una de las partes más importantes e interesantes del sistema, en la cual se escribieron las rutinas que inicializan el directorio y las tablas de páginas del kernel. El archivo modificado fue el `mmu.c`. Se implementó la función `mmu_inicializar_dir_kernel` cuyo código es el siguiente:

```
void mmu_inicializar_dir_kernel(){
    unsigned int * pageTable = (unsigned int *) 0x28000;
    unsigned int * tableDeDirecciones = (unsigned int *) 0x27000;
    *tableDeDirecciones = (unsigned int) pageTable | 3;
    tableDeDirecciones++;
    int i = 1 ;
    for(i = 1; i < 1024; i++){
        *tableDeDirecciones = 0;
        tableDeDirecciones++;
    }
    int j = 0 ;
    for(j = 0; j < 1024; j++){
        *pageTable = j*4096 | 3;
        pageTable++;
    }
}
```

Básicamente lo que se realizó es inicializar una PDE (directorio de páginas) en la dirección 0x27000 y completar su primera entrada con la dirección de una PTE (tabla de páginas) y con los atributos necesarios de supervisor para que sólo pueda ser accedida por el kernel. El resto de las 1024 entradas de la PDE fue completada con ceros ya que no van a ser utilizadas. La PTE fue completada para que mapee con identity mapping páginas de 4 KB, por lo que se completaron cada una de ellas con una dirección base múltiplo de 4096 y nuevamente con los atributos necesarios para que sólo puedan ser accedidas por el Kernel. La función `mmu_inicializar_dir_kernel` es llamada desde `kernel.asm` mediante un `call`.

c) Este otro punto es otro de los más importantes para el funcionamiento del sistema. Desde el kernel lo que se hizo fue inicializar el registro de control CR3. Como no es posible trabajar con el registro CR3 directamente, se copia en `eax`, el valor 0x27000 que es donde se definió en el punto anterior la PDE. Se copia el valor del registro `eax` a CR3 y se activa la paginación seteando el bit correspondiente en CR0 con un procedimiento similar al anterior. Para completar el proceso se hace un `call` a la función `mmu_inicializar` definida en `mmu.c` cuyo único propósito es setear una variable global que define la próxima página libre que puede ser utilizada por el sistema.

d) Ejercicio muy similar al de pintar pantalla sólo que además de un color se copian a memoria los valores ASCII del texto correspondiente al nombre de nuestro grupo.

1.4. Interrupciones

En esta parte, se completaron las interrupciones del reloj, la del teclado y la 0x66 que posteriormente será usada en el juego dado que la interrupción del reloj genera la CPU cada cierta cantidad de tiempo, recién en este momento se habilitan las interrupciones mediante la instrucción `STI`. El código de las interrupciones se encuentra en `isr.asm`

1.5. El retorno de la GDT y la aparición de la TSS

1.6. El Scheduler

Para realizar el scheduler, decidimos crear un struct tarea que contiene la posición en el mapa, su índice en la GDT, su `cr3`, si tiene prendido el bit de presente en la GDT

2. Conclusiones y trabajo futuro