DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming - Tierra Pirata

Organización del Computador II
Primer Cuatrimestre de 2015

Grupo Diablo II / PC

Integrante	LU	Correo electrónico
Ciruelos, Gonzalo Maddonni, Axel Thibeault, Gabriel	200 /14	axel.maddonni@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

Índice

1. Introducción

2. Desarrollo

2.1. Ejercicio 1

Inicialización de la GDT

Inicializamos la Tabla de Descriptores Globales con entradas para segmentos de código de nivel 0 y 3, otras para segmentos de datos de nivel 0 y 3, una para un segmento que describe el área de la pantalla de video, y la entrada correspondiente al segmento donde se guardará la tss de la tarea inicial. (Las entradas de gdt para las tss de las demás tareas son completadas al inicializarlas, como se explica en la sección correspondiente al Ejercicio 6).

Se utiliza desde el índice 8 por restricciones del trabajo práctico. Los segmentos de datos y códigos están organizados de tal forma que se superpongan direccionando los primeros 500MB de memoria (Sistema FLAT), utilizando bloques de 4K al setear el bit de granularidad en 1.

Los demás atributos fueron seteados de la siguiente manera:

Base y Límite: Como mencionamos anteriormente, los segmentos de código y datos están superpuestos. Comienzan en la dirección base 0x00000000, y el valor del límite 0x1F3FF corresponde la cantidad de bloques-1. Es decir, para cubrir 500MB se necesitan 128.000 bloques de 4K. El offset del último bloque es 127.999 (0x1F3FF en hexa). Con respecto al segmento de video, éste ocupa en memoria desde la posición 0xB8000 hasta la 0xC0000, es decir 32K de memoria, cuyo máximo offset o límite es el correspondiente al último byte (7999 = 0x7FFF). Para las tss, el límite es 0x68, pues miden 104 bytes cada una. Como base de la tarea inicial, seteamos la dirección 0x0000. ???????

Tipo: El tipo para los segmentos de código es 0x0A (executable, readable), mientras que para los de datos y video es 0x02 (readable, writable).

Sistema: El bit de system está seteado en 1 salvo en los segmentos correspondientes a las tss de las tareas, donde está activo bajo en 0 pues son potestad exclusiva del sistema operativo.

DPL: Los segmentos de datos y código de nivel 0 tienen DPL en 0x00, al igual que los segmentos de sistema y el de video, mientras que los de código y datos nivel 3 tienen DPL en 0x03.

Granularidad: El bit de G está activo sólo en los segmentos de datos y código ya que es necesario bloques de 4K para abarcar los 500MB.

P, L, D/B, AVL: Seteados en 1, 0, 1 y 0 respectivamente para todas las entradas.

Pasaje a Modo Protegido

Para pasar a modo protegido, completamos y cargamos la GDT usando la instrucción lgdt, que toma el descriptor de la GDT con el tamaño y la dirección de la misma, habilitamos A20 para habilitar el acceso a direcciones superiores a los 2^{20} bits, seteamos el bit de PE del registro CR0, y saltamos a 0x40:modoprotegido donde el 0x40 corresponde al Indice del segmento de código de nivel 0 (índice 8 en la gdt), corrido 3 ceros (estos ceros son los del TI y RPL).

Codigo 1: Pasaje a modo protegido

```
; Habilitar A20
call habilitar_A20

; Cargar la GDT
lgdt [GDT_DESC]      ; cargo la estructura que esta en gdt.c

; Setear el bit PE del registro CR0
mov eax, cr0
or eax, 1
mov cr0, eax

; Saltar a modo protegido
jmp 0x40:modoprotegido
```

Una vez trabajando en modo protegido, procedemos a establecer los selectores de segmentos de datos de nivel 0 (indice 9 en la gdt, corrido tres ceros correspondientes a los bits de TI y RPL), y el selector de segmento de video en fs (indice 12 en la gdt). Luego establecemos la base de la pila en la dirección 0x27000.

Codigo 2: Pasaje a modo protegido

```
modoprotegido:

; Establecer selectores de segmentos
xor eax, eax
mov ax,  1001000b
mov ds, ax
mov es, ax
mov gs, ax
mov ss, ax

mov ax, 1100000b
mov fs, ax

; Establecer la base de la pila
mov ebp, 0x27000
mov esp, 0x27000
```

Inicialización de la Pantalla

Para inicializar la pantalla llamamos a la función de screen.h *screen_inicializar*, que se encarga de pintar la pantalla con el mapa, las barras para los jugadores, e inicializar los slots vacios y puntos en 0 de cada jugador, utilizando las funciones *screen_pintar_rect* para pintar un rectangulo de color, *print_dec* para imprimir los puntos, y *screen_pintar* para imprimir caracteres.

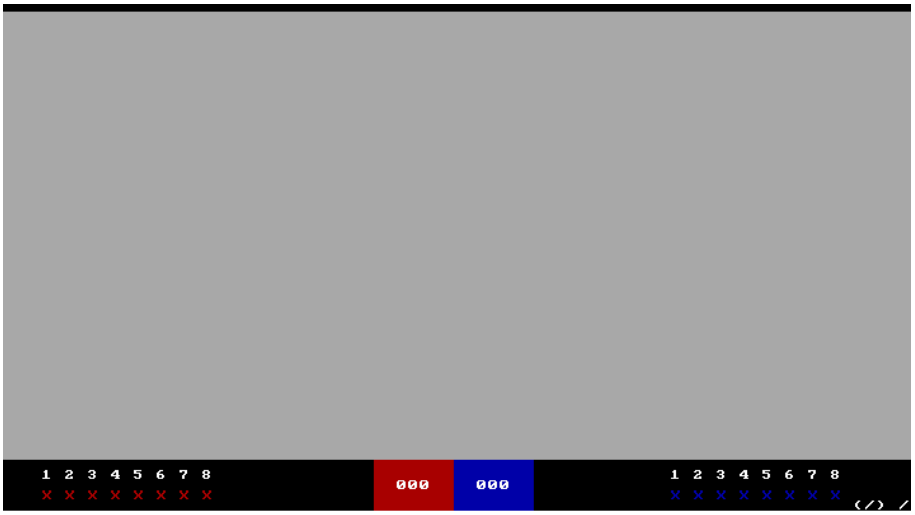

Figura 1: Pantalla Inicial.

2.2. Ejercicio 2

Inicialización de la IDT

Para inicializar la IDT se llama a una función en lenguaje C, ”void idt_inicializar(void)”. La IDT se representa mediante un arrelgo (de tamaño 255) de ïdt_entry”(esta estructura fue definida como lo muestra el extracto de Código 3, según las especificaciones dadas en el manual de Intel). Esta función utiliza un macro (que se encuentra en el extracto de Código 4) para inicializar cada una de las entradas necesarias de la IDT.

El macro define el offset como los 16 bits menos y más significativos (ya que éste se encuentra separado en dos campos de 16 bits cada uno) de la dirección de la tarea de atención de la interrupción correspondiente. El selector de segmento lo define como 0x0040, que es 0x8 (el indice del segmento de codigo de nivel de privilegio 0 en la GDT) shifteado 3 bits a la izquierda. Los atributos los define como 0x8e00, que representan un segmento presente (P = 1), un nivel de privilegio de 0 (DPL = 00), un tamaño de Gate de 32 bits (bits 8 a 12 de la Interrupt Gate = 0b01110), y los 7 bits restantes en 0.

La rutina de atención de cada interrupción se genera a partir de un macro (que se encuentra en el extracto de Código 5) que imprime el código de error correspondiente a pantalla y luego queda en un loop infinito.

La IDT ya inicializada se puede acceder tras ejecutar la instrucción $lidt[IDT\_DESC]$. El descriptor de la IDT, ÏDT_DESC.[es] una estructura (definida como se ve en el extracto de código **??**) que contiene el tamaño de la IDT y su dirección en memoria.

Codigo 3: Estructura de idt_entry

```
typedef struct str_idt_entry_fld {
    unsigned short offset_0_15;
    unsigned short segsel;
    unsigned short attr;
    unsigned short offset_16_31;
} __attribute__((__packed__, aligned (8))) idt_entry;
```

Codigo 4: Codigo del macro utilizado para inicializar la IDT

```
#define IDT_ENTRY(numero)
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero)
        & (unsigned int) 0xFFFF);
    idt[numero].segsel = (unsigned short) 0x0040;
    idt[numero].attr = (unsigned short) 0x8e00;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero)
         >> 16 & (unsigned int) 0xFFFF);
```

Codigo 5: Codigo del macro utilizado para la rutina de atencion de interrupciones

```
_isr%1:
    mov eax, %1
    push dword 0x0f0f
    push dword 0
    push dword 0
    push MENSAJE_ERROR_%1
    call print

    jmp \$
```

Codigo 6: Estructura de IDT_Desc

```
typedef struct str_idt_descriptor {
    unsigned short  idt_length;
    unsigned int    idt_addr;
} __attribute__((__packed__)) idt_descriptor;
```

2.3. Ejercicio 3

Para inicializar el directorio del kernel, lo que hacemos es, en la primera posicion declarar la tabla de kernel (que será identity mapping), y luego ponemos todo el resto de directorio en 0 (bit de presente en 0, que indica que esas entradas no direccionan nada).

Luego inicializamos la tabla de kernel, que está en identity mapping, como dijimos anteriormente. Eso es bastante fácil, ya que podemos usar la misma variable para iterar y para decir la dirección física a la que direccionará una dirección virtual.

Para pintar la pantalla usamos las funciones que nos permiten pintar rectángulos, que no necesitan explicación dado que son muy simples.

2.4. Ejercicio 4

Inicialización de la MMU

Para administrar la memoria en el área libre contamos con un contador de páginas inicializado en la dirección 0x00100000. A medida que el sistema necesita una página, éste contador se incrementa en 4K, como muestra la siguiente implementación:

Codigo 7: Contador de Páginas Libres

```
void * siguiente_libre;

void inicializar_mmu()
{
  siguiente_libre = (void *) PAGE_COUNTER_INIT;
}

void * dar_siguiente()
{
    uint i;
    for(i = 0; i<1024; i++) ((pde *) siguiente_libre)[i].present = 0;
    siguiente_libre += 0x1000;
    return siguiente_libre - 0x1000;
}
```

Al crear una página, recorremos todas entradas de la tabla seteando el bit de presente en 0 (sea ésta un directorio o tabla de páginas). Para simplificar la manipulación en el código de las pde y pte creamos dos estructuras en C correspondientes a las ya mencionadas:

Codigo 8: struct Page Directory Entry

```
typedef struct pde_t {
    unsigned char present:1;
    unsigned char read_write:1;
    unsigned char user_supervisor:1;
    unsigned char page_level_write_through:1;
    unsigned char page_level_cache_disable:1;
    unsigned char accessed:1;
    unsigned char reserved:1;
    unsigned char page_size:1;
    unsigned char global:1;
    unsigned char available_9_11:3;
    unsigned int  base_address:20;
} __attribute__((__packed__, aligned (4))) pde;
```

Codigo 9: struct Page Table Entry

```
typedef struct pte_t {
    unsigned char present:1;
    unsigned char read_write:1;
    unsigned char user_supervisor:1;
    unsigned char page_level_write_through:1;
    unsigned char page_level_cache_disable:1;
    unsigned char accessed:1;
    unsigned char dirty:1;
```

```
    unsigned char page_table_attribute_index:1;
    unsigned char global:1;
    unsigned char available_9_11:3;
    unsigned int  base_address:20;
} __attribute__((__packed__, aligned (4))) pte;
```

Inicialización de directorios y tablas para Tareas Pirata
Mapeo y Desmapeo de Páginas
Testeo de Paginación (item no implementado en la sol final)

2.5. Ejercicio 5

Interrupción de Reloj
Interrupción de Teclado
Interrupción de sistema (0x46)

2.6. Ejercicio 6

2.7. Ejercicio 7

2.8. Ejercicio 8

3. Conclusiones