



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Programacion SIMD

Organización del Computador II
Segundo Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Leandro Raffo		
Maximiliano Fernández Wortman		
Uriel Rozenberg		



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

Ejercicio 1.

Básicamente armamos la tabla de gdt, empezando en la posición 8 del array, con 4 segmentos, dos de código nivel 0 y 3 y dos de datos nivel 0 y nivel 3, llamados respectivamente:

```
GDT_IDX_CS_CERO_DESC
GDT_IDX_CS_TRES_DESC
GDT_IDX_DS_CERO_DESC
GDT_IDX_DS_TRES_DESC
```

Excepto el dpl y el tipo todos estos segmentos tienen un límite de 0x1E400, con base en 0x0000, ya que esto nos da 0x1F400 que son 500mb de memoria, esto es porque tenemos la granularidad en 1 es decir contamos de a 4k.

Para pasar a modo protegido usamos el siguiente código.

```
    jmp 0x40:modoprotegido
BITS 32
modoprotegido:
    mov eax, 0x50
    mov ss, ax
    mov ds, ax
    mov gs, ax
    mov fs, ax
    mov es, ax
    mov ebp, 0x1337
    mov esp, 0x28000
```

Hacemos un far jump con el offset para CS tenga el valor de nuestro código en nivel 0 que es $8 \ll 3 = 0x40$ y ponemos el comando BITS 32 para que el ensamblador sepa que las próximas instrucciones sean generadas para un procesador de 32 bits, seteamos todos los registros de segmentos con el de datos de nivel 0 ($10 \ll 3 = 0x50$) y definimos el stack esp en 0x28000 como pide el enunciado.

Para el punto siguiente usamos la función call screen_inicializar para pintar la pantalla.

Ejercicio 2.

Tenemos que definir las interrupciones, para esto usamos el macro IDT_ENTRY, este se encuentra en idt.c y lo llamamos con los siguientes valores adentro de la función idt_inicializar, donde el primer valor es el número de la interrupción y el segundo el del dpl:

```
IDT_ENTRY(0, 0);    /* Divide error          */
IDT_ENTRY(2, 0);    /* NMI Interrupt          */
IDT_ENTRY(3, 0);    /* Breakpoint - INT 3     */
IDT_ENTRY(4, 0);    /* Overflow               */
IDT_ENTRY(5, 0);    /* BOUND Range Exceeded   */
IDT_ENTRY(6, 0);    /* Invalid opcode         */
IDT_ENTRY(7, 0);    /* Device Not Available    */
IDT_ENTRY(8, 0);    /* Double Fault           */
IDT_ENTRY(9, 0);    /* Coprocessor segment overrun */
IDT_ENTRY(10, 0);   /* Invalid TSS            */
IDT_ENTRY(11, 0);   /* Segment not present    */
IDT_ENTRY(12, 0);   /* Stack-Segment Fault    */
IDT_ENTRY(13, 0);   /* General Protection     */
IDT_ENTRY(14, 0);   /* Page Fault             */
IDT_ENTRY(16, 0);   /* x87 FPU Floating-Point Error */
IDT_ENTRY(17, 0);   /* Alignment Check        */
IDT_ENTRY(18, 0);   /* Machine Check          */
IDT_ENTRY(19, 0);   /* SIMD Floating-Point Exception */
```

Una vez echo esto en isr.asm usamos el macro

```
%macro ISR 1
global _isr%1

_isr%1:
    mov eax, %1
    jmp $

%endmacro
```

Para definir las 18 interrupciones.

```
ISR 0
.
.
.
ISR 19
```

Luego de esto en kernel.asm pusimos el siguiente codigo:

```
call idt_inicializar
lidt [IDT_DESC] ; igual que con la gdt
```

Es decir inicializamos la idt y la cargamos a su registro correspondiente. Para probar que esto andaba hicimos una división por zero:

```
mov eax, 0
mov ebx, 0
div ebx
```

Lo cual hizo saltar la excepción 0, de divide error.

Ejercicio 3.

Para este punto programamos la función `mmu_mapear_pagina`:

```
void mmu_mapear_pagina(uint virtual, uint cr3, uint fisica, uint attrs) {
    cr3 = cr3 & 0xFFFFF000;
    uint pduint = cr3;
    uint posicion_DR = (virtual >> 22) & 0x3FF;
    pduint = pduint + (posicion_DR * 4);
    page_directory *pd = (page_directory *)pduint;
```

Esta función recibe una dirección virtual, un cr3, una dirección física y los atributos. Primero conseguimos la dirección de cr3 por medio de un and contra 0xFFFFF000 y la guardamos en pduint. Luego shifteamos la dirección virtual a la derecha 22 bits para tener la posición de la entrada del directorio de páginas. Una vez que tenemos este valor se lo sumamos a pduint y casteamos el resultado a `page_directory *`, donde `page_directory` es el siguiente struct:

```
typedef struct page_directory {
    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_disabled:1;
    unsigned char    accessed:1;
    unsigned char    ignored:1;
    unsigned char    page_size:1;
    unsigned char    global:1;
    unsigned char    available_11_9:3;
    unsigned int     page_base_address_31_12:20;
} __attribute__((__packed__, aligned (4))) page_directory;
```

Luego nos fijamos si está presente, de no estarlo le pedimos una página física libre a `mmu_proxima_fisica_libre` y la inicializamos con la función `mmu_inicializar_page_directory`:

```
void mmu_inicializar_page_directory(page_directory * dir, uint addr, uint attrs) {
    dir->present = attrs & 0x1;
    dir->read_write = (attrs >> 1) & 0x1;
    dir->user_supervisor = (attrs >> 2) & 0x1;
    dir->page_level_write_through = (attrs >> 3) & 0x1;
    dir->page_level_cache_disabled = (attrs >> 4) & 0x1;
    dir->accessed = (attrs >> 5) & 0x1;
    dir->ignored = (attrs >> 6) & 0x1;
    dir->page_size = (attrs >> 7) & 0x1;
    dir->global = (attrs >> 8) & 0x1;
    dir->available_11_9 = (attrs >> 9) & 0x3;
    dir->page_base_address_31_12 = addr >> 12;
}
```

Luego usando la función `mmu_inicializar_page_table`

```
void mmu_inicializar_page_table(page_table *tab, uint addr, uint attrs) {
    tab->present = attrs & 0x1;
    tab->read_write = (attrs >> 1) & 0x1;
    tab->user_supervisor = (attrs >> 2) & 0x1;
    tab->page_level_write_through = (attrs >> 3) & 0x1;
    tab->page_level_cache_disabled = (attrs >> 4) & 0x1;
    tab->accessed = (attrs >> 5) & 0x1;
    tab->dirty_bit = (attrs >> 6) & 0x1;
    tab->page_table_attr_indx = (attrs >> 7) & 0x1;
    tab->global = (attrs >> 8) & 0x1;
    tab->available_11_9 = (attrs >> 9) & 0x3;
    tab->page_base_address_31_12 = addr >> 12;
}
```

Ponemos todas las entradas de la page table a la que apunta, en no presente y con atributos en 0.

```
if (pd->present == NULL) {
    uint proxima_pag = mmu_proxima_pagina_fisica_libre();
    mmu_inicializar_page_directory(pd, proxima_pag, 0x3);
    int tab_c = proxima_pag;
    for (; tab_c < proxima_pag + 0x1000; tab_c += 0x4)
        mmu_inicializar_page_table((page_table *)tab_c, 0, 0);
}
```

Por último shifteamos la dirección virtual a la derecha 12 bits y lo guardamos en `posicion_DT` esto nos va a servir para obtener la entrada en la tabla de páginas, luego shifteamos la entrada del directorio de página 12 bits a la izquierda para obtener la dirección base (múltiplo de 4k) y se lo sumamos a `posicion_DT`, por último lo casteamos a tipo `page_table` e inicializamos la página con la dirección física y los atributos que nos habian pasado usando de vuelta `mmu_inicializar_page_table`.

```
uint posicion_DT = (virtual >> 12) & 0x3FF;
uint add = pd->page_base_address_31_12 << 12;
page_table *pt = (page_table *) (add + (posicion_DT * 4));
mmu_inicializar_page_table(pt, fisica, attrs);
}
```

Ejercicio 4.

Ejercicio 5.

Ejercicio 6.

Ejercicio 7.