DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

27 de octubre de 2015 — Organizacion del computador II

Te voy a dar un Byte

Integrante	LU	Correo electrónico
Gonzalez Benitez, Albertito Juan	324/14	gonzalezjuan.ab@gmail.com
Lew, Axel Ariel	225/14	axel.lew@hotmail.com
Noli Villar, Juan Ignacio	174/14	juaninv@outlook.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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

0.1. Introduccion

En este trabajo practico procederemos a realizar un sistema operativo que pueda correr un juego.

1. Ejercicio 1:

1.1. Introducción:

En este ejercicio vamos a realizar la Tabla de Descriptores Globales (*GDT*). Se pide que realizemos lo siguiente :

a) Que la tabla *GDT* tenga 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 500MB de memoria. Por último se pide no usar las primeras siete posiciones de la *GDT*, ya que se consideran utilizadas.

b) Pasar a modo protegido y setear la pila del kernel en la dirección 0x27000.

c) Agregar a la *GDT* un segmento adicional y escribir una rutina qe utilice este nuevo segmento para pintar la esquina superior izquierda de la pantalla.

d) Limpiar la pantalla y pintar el área del mapa (sugerido el color gris) junto con las barras inferiores para los jugadores.

1.2. Ítem a): Setear la *GDT*

Para este ítem completamos el archivo *GDT*.c proporcionado por la catedra. En el mismo la *GDT* es representada mediante un array de 30 posiciones. Cada posicion tiene la siguiente estructura.

```
[GDT_IDX_NULL_DESC] = (gdt_entry) {
    (unsigned short)   0x0000,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,       /* base[23:16]  */
    (unsigned char)    0x00,       /* type         */
    (unsigned char)    0x00,       /* s            */
    (unsigned char)    0x00,       /* dpl          */
    (unsigned char)    0x00,       /* p            */
    (unsigned char)    0x00,       /* limit[16:19] */
    (unsigned char)    0x00,       /* avl          */
    (unsigned char)    0x00,       /* l            */
    (unsigned char)    0x00,       /* db           */
    (unsigned char)    0x00,       /* g            */
    (unsigned char)    0x00,       /* base[31:24]  */
},
```

Figura 1: Este descriptor corresponde a la primer entrada de la *GDT*

Como la primer posicion de la tabla *GDT* debe ser corresponder a una entrada nula, llenamos la primer posicion como muestra la imagen debajo.

```
[GDT_IDX_NULL_DESC] = (gdt_entry) {
    (unsigned short)   0x0000,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,        /* base[23:16]  */
    (unsigned char)    0x00,        /* type         */
    (unsigned char)    0x00,        /* s            */
    (unsigned char)    0x00,        /* dpl          */
    (unsigned char)    0x00,        /* p            */
    (unsigned char)    0x00,        /* limit[16:19] */
    (unsigned char)    0x00,        /* avl          */
    (unsigned char)    0x00,        /* l            */
    (unsigned char)    0x00,        /* db           */
    (unsigned char)    0x00,        /* g            */
    (unsigned char)    0x00,        /* base[31:24]  */
},
```

Figura 2: Este descriptor corresponde a la primer entrada de la *GDT*

Luego, creamos los 4 segmentos que se piden a partir de la posicion 8 de la *GDT*, ya que por enunciado, no se deben tocar las primeras 7 posiciones de la table de descriptores. Mostramos en las imagenes de abajo como creamos un descriptor de datos y otro de codigos.

```
[GDT_IDX_NULL_DESC+8] = (gdt_entry) {
    (unsigned short)   0xF400,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,        /* base[23:16]  */
    (unsigned char)    0x0A,        /* type         */
    (unsigned char)    0x01,        /* s            */
    (unsigned char)    0x00,        /* dpl          */
    (unsigned char)    0x01,        /* p            */
    (unsigned char)    0x01,        /* limit[16:19] */
    (unsigned char)    0x00,        /* avl          */
    (unsigned char)    0x00,        /* l            */
    (unsigned char)    0x01,        /* db           */
    (unsigned char)    0x01,        /* g            */
    (unsigned char)    0x00,        /* base[31:24]  */
},
```

Figura 3: Este descriptor corresponde al segmento de datos de nivel 0

```
[GDT_IDX_NULL_DESC+9] = (gdt_entry) {
    (unsigned short)   0xF400,      /* limit[0:15]  */
    (unsigned short)   0x0000,      /* base[0:15]   */
    (unsigned char)    0x00,        /* base[23:16]  */
    (unsigned char)    0x02,        /* type         */
    (unsigned char)    0x01,        /* s            */
    (unsigned char)    0x00,        /* dpl          */
    (unsigned char)    0x01,        /* p            */
    (unsigned char)    0x01,        /* limit[16:19] */
    (unsigned char)    0x00,        /* avl          */
    (unsigned char)    0x00,        /* l            */
    (unsigned char)    0x01,        /* db           */
    (unsigned char)    0x01,        /* g            */
    (unsigned char)    0x00,        /* base[31:24]  */
},
```

Figura 4: Este descriptor corresponde al segmento de código de nivel 0

Los otros dos que faltan son exactamente iguales, solo que en la linea correspondiente al nivel (dpl) ponemos 0x03, ya que corresponde al nivel 3 de prioridad.

Los descriptores de segmentos tienen la siguiente forma:

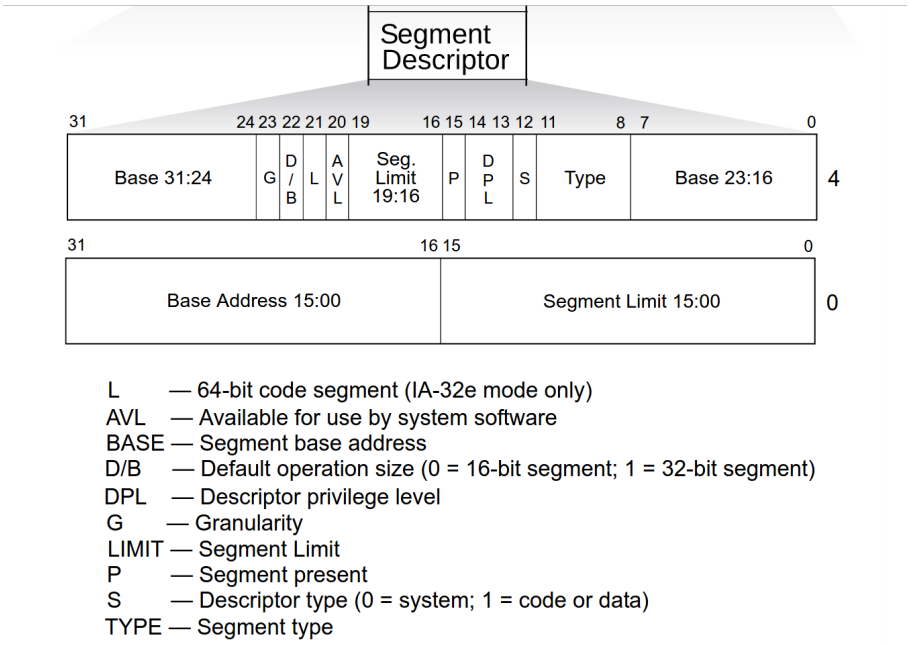

Figura 5: Este descriptor corresponde a la primer entrada de la *GDT*

Completamos nuestros descriptores como marcan las figuras 3 y 4 de esa forma porque:

Base: En el tp se pide que los descriptores direcciones los primeros 500mb de memoria. Por ende la base corresponde a la dirección 0x00000000.

G: Para poder direccionar 500mb, no nos alcanza la cantidad de bits que hay para el limite, por ende necesitamos activar la granularidad para poder abarcar más memoria, ya que cuando esta activada la posicion que indica el limite se multiplica por 4kb.

Límite: Como esta activada la granularidad, podemos abarcar 500mb de memoria, el limite correspondiente a 500mb con la granularidad activada es 0x0F400.

Type: Aqui se indica si el descriptor es de código/datos, a los correspondientes a datos les pusimos que eran de tipo 0x02 (segmento de datos de escritura/lectura) y a los de código que eran de tipo 0x0A (segmentdo de código de escritura/lectura).

S: Con este bit se decide si es un segmento de sistema (s=0) o si son de código/data (s=1), por ende a este bit le corresponde un 0x1.

Dpl: En esta seccion se declara el privilegio del segmento, a los que eran de nivel 0 les corresponde un 0x00 y a los de nivel 3 un 0x03

P: Este es el bit de present. Cuando es '1' el segmento correspondiente esta presente en la memoria RAM. Si es '0', el segmento esta en la memoria virtual. Por ende lo seteamos en 1.

Avl: Es el bit correspondiente a Available. Como no lo vamos a tener en cuenta lo dejamos en 0.

L: Indica si el código es de 64bits o de 32. Como trabajamos en 32bits dejamos este bit en 0.

D/B: Este bit define el tamaño de las operaciones en las que va a trabajar el procesador. De nuevo, como nos encontramos trabajando en 32bits, el tamaño de las operaciones debe ser de 32, por eso lo seteamos en 1.

1.3. Ítem b): Pasar a modo protegido y setear la pila

Para pasar a modo protegido realizamos los siguientes pasos:

a) Deshabilitamos las interrupciones, para eso utilizamos la instrucción CLI.

b) Cargamos el registro *GDTR* con la dirección base de la *GDT* utilizando la instrucción *LGDT* de esta manera:

```
lgdt [GDT DESC]
```

Donde GDT DESC es el descriptor de la tabla.

c) Seteamos el bit *PE* (BIT 0) del registro *CR0* en 1 para pasar a modo protegido. Procedemos a hacer esto con las intrucciones:

```
mov eax, cr0
or eax, 1
mov cr0, eax
```

d) Realizamos un far jump para cargar en el registro *CS* la dirección donde esta el segmento de código. Para esto utilizamos la instruccion:

```
jmp 0X40:modoprotegido
```

Donde 0x40 es la dirección donde en nuestra *GDT* compienza el segmento de código y modoprotegido es una etiqueta que se encuentra imediatamente debajo de este JMP.

e) Cargamos los registros de segmento de la siguiente manera.

```
mov ax, 0x48
mov ds, ax
mov ax, 0x48
mov ss, ax
```

Ahora que ya nos encontramos con el procezador trabajando en modo protegido, procedemos a setear la pila en la dirección 0x27000. Para eso seteamos los registros ebp y esp en la dirección 0x27000 con las siguientes instrucciones:

```
mov ebp, 0x27000
mov esp, 0x27000
```

1.4. Ítem c): Agregar a la *GDT* un segmento adicional y utilizarlo como memoria de vídeo

Para este ítem, agregamos a la *GDT* el siguiente segmento, el cual direcciona a la memoria de vídeo utilizada por la pantalla (Desde 0xB8000 a 0XC000).

```
// VIDEO
[GDT_IDX_NULL_DESC+12] = (gdt_entry) {
   (unsigned short)   0x8000,      /* limit[0:15]  */
   (unsigned short)   0x8000,      /* base[0:15]   */
   (unsigned char)    0x0B,       /* base[23:16]  */
   (unsigned char)    0x02,       /* type         */
   (unsigned char)    0x01,       /* s            */
   (unsigned char)    0x00,       /* dpl          */
   (unsigned char)    0x01,       /* p            */
   (unsigned char)    0x02,       /* limit[16:19] */
   (unsigned char)    0x00,       /* avl          */
   (unsigned char)    0x00,       /* l            */
   (unsigned char)    0x01,       /* db           */
   (unsigned char)    0x00,       /* g            */
   (unsigned char)    0x00,       /* base[31:24]  */
},
```

Figura 6:

Luego cargamos en un registro de segmentos, en este caso *FS*, la posicion del segmento anterior. Por ultimo con las siguiente instruccion podemos poner en la esquina superior izquierda de la pantalla lo que querramos

mov word[fs:0x00], *aimprimir*

Donde *aimprimir* es algo de tamaño word, y su valor es lo que aparece en la esquina superior izquierda de la pantalla. Esto ocurre porque se carga el semento que comienza en la dirección a que la dirección 0xB8000 y se le suma el offset 0 y esta es la primer dirección de la memoria de vídeo utilizada por la pantalla.

Si por ejemplo escribimos la siguiente línea:

mov word[fs:0x00], 0xdb00

Obtenemos este resultado, que es lo que pedía el ejercicio.

Figura 7:

1.5. Ítem d): Limpiar la pantalla y pintar el área del mapa

En este punto creamos una funcion auxiliar en C para limpiar la pantalla. La función pinta la pantalla de gris y es la siguiente:

```
void aux_limpiarPantalla(){
  int i = 1;
  while (i<45){
    int j = 0;
    while(j<80){
      p[i][j].c = 219;
      p[i][j].a = 7;
      j++;
    }
    i++;
  }
```

Figura 8:

Esta función la escribimos en *screen.c* y utiliza la matriz P creada por la catedra para acceder a las posiciones de la pantalla. Estos son los resultados luego de utilizar nuestra función.

Figura 9:

Para pintar el mapa utilizamos la funcion dada por la catedra *screen_inicializar*. Para utilizar estas funciones, escribimos en *kernel.asm* las siguientes lineas con el siguiente resultado:

call aux_limpiarPantalla
call screen_inicializar

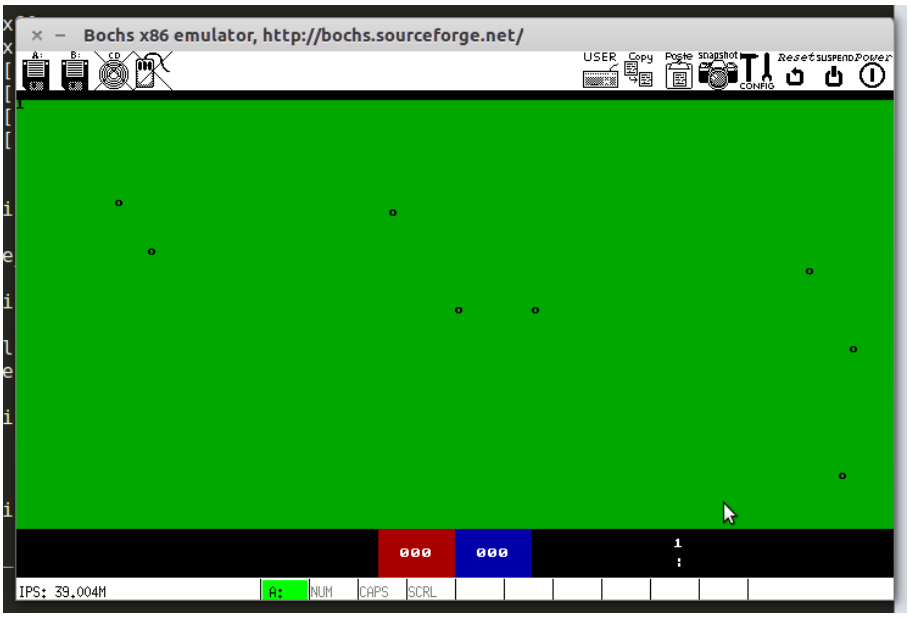
Bochs x86 emulator, http://bochs.sourceforge.net/
USER
Copy
Paste
snapshot
CONFIG
Reset
suspend
Power
000
000
1
:
IPS: 39.004M
A:
NUM
CAPS
SCRL

Figura 10:

2. Ejercicio 2:

2.1. Introducción:

En este ejercicio vamos a setear las interrupciones basicas, osea las reservadas para el procesador. Para esto completaremos la *IDT*. Se pide que realizemos lo siguiente :

a) Completar la *IDT* de tal forma que indique por pantalla el problema que se produjo y que interrumpa la ejecución.

b) Cargar la *IDT* y probarla.

2.2. Ítem a): Setear la *IDT*

Para esto utilizamos los siguientes archivos dados por la catedra *idt.c*, *isr.h*, *isr.asm* y los completamos.

2.2.1. *Idt.c:*

En este archivo se encuentra la función *IDT_ENTRY(número, dpl)* la cual recive el numero de la interrupción y su prioridad. La misma se encontraba incompleta y necesitaba ser completada con la información necesaria. La llenamos agregandole el segmento correspondiente y los atributos correpondientes. El segmento que le correpondia se ubicaba en la direccion 0x40 (explicado en el ejercicio 1) y los atributos son los correpondientes a el valor 0x8700, ya que el formato de los atributos de una interrupcion debe ser de la siguiente forma:

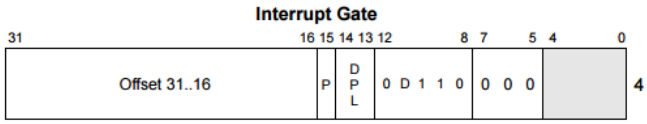

Figura 11: Formato de los atributos de una interrupcion

Entonces como 0x8700 representa en binario 1000 0111 0000 0000, tiene el formato que queremos. Ya que P debe ser 1, DPL debe ser 000, luego sigue un 0, luego D (que en este caso es 1 ya que estamos trabajando en 32bits) luego siguen dos unos y por último ocho ceros, lo cual forma el número que queremos. De esta forma nos queda la funcion *IDT_ENTRY* de la siguiente manera:

```
#define IDT_ENTRY(numero, dpl)                                                  \
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);      \
    idt[numero].segsel = (unsigned short) 0x40;                                 \
    idt[numero].attr = (unsigned short) 0x8700 | (((unsigned short)(dpl & 0x3)) << 13);                  \
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xFFFF);
```

Figura 12: Formato de los atributos de una interrupcion

Por último, creamos las 19 interrupciones correspondientes utilizando *IDT_ENTRY* .

```
void idt_inicializar() {
    // Excepciones
    IDT_ENTRY(0, 0);
    IDT_ENTRY(1, 0);
    IDT_ENTRY(2, 0);
    IDT_ENTRY(3, 0);
    IDT_ENTRY(4, 0);
    IDT_ENTRY(5, 0);
    IDT_ENTRY(6, 0);
    IDT_ENTRY(7, 0);
    IDT_ENTRY(8, 0);
    IDT_ENTRY(9, 0);
    IDT_ENTRY(10, 0);
    IDT_ENTRY(11, 0);
    IDT_ENTRY(12, 0);
    IDT_ENTRY(13, 0);
    IDT_ENTRY(14, 0);
    IDT_ENTRY(15, 0);
    IDT_ENTRY(16, 0);
    IDT_ENTRY(17, 0);
    IDT_ENTRY(18, 0);
    IDT_ENTRY(19, 0);
```

Figura 13: Formato de los atributos de una interrupcion

2.2.2. *Isr.h:*

En este archivo tuvimos que declarar las siguientes funciones:

```
#ifndef __ISR_H__
#define __ISR_H__

void _isr0();
void _isr1();
void _isr2();
void _isr3();
void _isr4();
void _isr5();
void _isr6();
void _isr7();
void _isr8();
void _isr9();
void _isr10();
void _isr11();
void _isr12();
void _isr13();
void _isr14();
void _isr15();
void _isr16();
void _isr17();
void _isr18();
void _isr19();

#endif /* !__ISR_H__ */
```

Figura 14: Formato de los atributos de una interrupcion

Sino, la función del archivo anterior, *IDT_ENTRY*, no compilaba, pues no las encontraba. No es necesario que estas funciones hagan algo, ya que en realidad *IDT_ENTRY* las utilizaba como macro.

2.2.3. *Isr.asm:*

Por último, en este archivo, atendemos las interrupciones con sus correspondientes rutinas, como en el enunciado solo se pide que se interrumpa la ejecucion del programa y se muestre por pantalla la iterrupcion que genero el problema, solamente utilizamos una funcion macro dada por la catedra que simplemente muestra dicho mensaje y ejecuta un loop infinito.

La macro es la siguiente:

```
exception1 db    'Divide Error', 0


%macro ISR 1
global _isr%1

_isr%1:
   mov eax, %1
   push 0xf
   push 0
   push 0
   push exception%1
   call print
   sub esp, 4
   sub esp, 4
   sub esp, 4
   sub esp, 4
   jmp $

%endmacro
```

Figura 15: Macro para atender interrupciones

La macro de la imágen anterior reemplaza %1 por el primer parametro que recibe. Este parametro es el número correpsondiente a la interrupción, por ende cuando llegue la interrupucion "x", se pushea el mensaje exception"x" y print muestra por pantalla el mensaje correspondiente a la etiqueta exception"x", para luego quedarse saltando infinitamente y por lo tanto interrupiendo la ejecucion del programa. En la imagen se muestra un ejemplo del mensaje que se mostraria si se produjera la interrupcion 1 , correspondiente a la division por cero.

Por ultimo escribimos la rutina de atencion de interrupciones, la cual simplemente llama a la macro anterior, tomando como parámetro el número de la interrupcion.

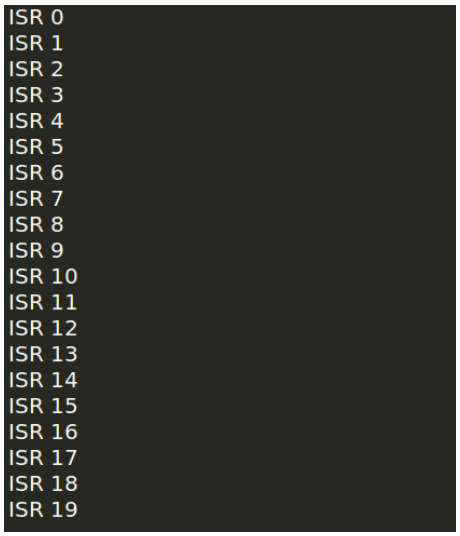

Figura 16: Rutina de atencion de interrrupciones.

2.3. Ítem b): Cargar y probar *IDT*

Para cargar la *IDT* y probarla, agregamos las siguientes lineas al *kernel*:

```
lidt [IDT_DESC]
    mov eax, 0
    mov ecx, 1
    div ecx
```

Obteniendo el resultado esperado:

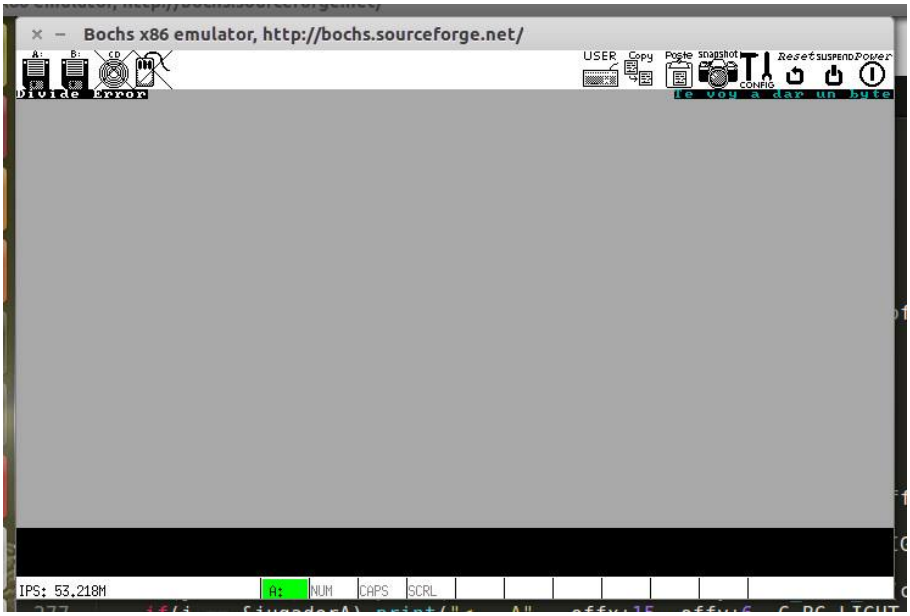

Figura 17: Rutina de atencion de interrrupciones.

3. Ejercicio 3: