DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Batalla Bytal

Organización del Computador II

Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Carlos Daniel Núñez Morales	732/08	cdani.nm@gmail.com
Alejo Antonio Salvador	467/15	alelucmdp@hotmail.com
Guido Joaquin Tamborindeguy	584/13	guido@tamborindeguy.com.ar
Pedro Facundo Tamborindeguy	627/11	pftambo@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

En este trabajo tenemos como objetivo mediante el programa Bochs, el cual lo utilizaremos como entorno de pruebas, poder aplicar los conceptos de System Programming vistos en clases teoricas y practicas.

El sistema deberá poder correr hasta ocho tareas concurrentemente a nivel de usuario, para esto tendremos que poder iniciar una computadora, es decir, 'bootear' y ceder el control al kernel, que se ocupará de ejecutar las dichas tareas. El código del boot-sector, como así todo el esquema de trabajo para armar el kernel y correr tareas, es provisto por la cátedra.

En este informe, describiremos detalladamente como realizamos la implementación de este sistema paso a paso, es decir, como hicimos para completar cada ejercicio por separado, llegando así a la versión final.

2. Ejercicio 1

En este ejercicio creamos dos descriptores de datos y dos de código, e inicializamos la GDT con ellos, también le agregamos un descriptor para el segmento de video. Luego pasamos a modo protegido, seteamos la pila en la dirección 0X27000 y finalmente probamos que el descriptor de video sea correcto usandolo para escribir en pantalla.

a) En este item lo que hicimos fue inicializar la GDT. Por restricción del trabajo práctico las 17 primeras posiciones se consideran usadas. En ese sentido cargaremos los descriptores a partir del índice 18 de la siguiente manera: en el indice 18 y 19 descriptores de codigo para kernel y usuario, en el 20 y 21 descriptores de datos para kernel y usuario, el 22 el descriptor de video. Para realizar esto usamos el arreglo de descriptores (GDT) provisto por la cátedra cuyo código se encuentra en $gdt.h$ y $gdt.c$, seteamos los valores adecuadamente para las posiciones del 18 al 22 (para indexar el arreglo usamos constantes definidas en $defines.c$).

```
//la entrada 18 de la gdt sera el segmento de codigo de nivel 0
[GDT_IDX_COD_0] = (gdt_entry){
(unsigned short)  0xFFFF, // limit[0:15] 1,75Gb=1835008/4Kb = 458752 Kb => Offset = 458751 = 0x6FFFF
(unsigned short)  0x0000, // base[0:15]
(unsigned char)   0x00,   // base[23:16]
(unsigned char)   0x0A,   // type        execute/read
(unsigned char)   0x01,   // s          En este caso es 'code'
(unsigned char)   0x00,   // dpl        level 0
(unsigned char)   0x01,   // p
(unsigned char)   0x06,   // limit[16:19]
(unsigned char)   0x00,   // avl
(unsigned char)   0x00,   // l
(unsigned char)   0x01,   // db         db = 1 para que sea Kilobyte
(unsigned char)   0x01,   // g
(unsigned char)   0x00,   // base[31:24]
    },
//le sigue el descriptor de segmento de codigo de nivel 3, que será igual a excepcion del dpl
//luego los dos descriptores de datos (nivel 0 y 3), direccion base y limite son exactamente igual.
```

Aclaración: para el descriptor de segmento de nivel 0 ponemos el campo tipo en 0xA (1010) y el bit S en 1 ya que no es de sistema (es de código o datos). El bit de presente lo ponemos en 1 porque el segmento esta presente. Como trabajamos con 32 bits ponemos d/b en 1 y l en 0.
El segmento de datos es igual que el segmento de código, salvo por el tipo, que en este caso es 0x2 (0010) porque es un segmento de datos que se puede leer y escribir.

b) Pasando a modo protegido y seteando la pila. Para eso, una vez cargada la gdt, deshabilitamos las interrupciones, cargamos el registro $GDTR$, seteamos el bit PE del registro $CR0$, se cragan los registros de segmento

```
; habilitar A20
  call habilitar_A20
  ; cargar la GDT
  lgdt [GDT_DESC]
  ; setear el bit PE del registro CR0
  mov eax, cr0
  or eax, 1
  mov cr0, eax
  jmp 0x90:modoprotegido; cs => code level 0 segment, index: 18
  ; pasar a modo protegido
```

```
BITS 32
    modoprotegido:
    ; acomodar los segmentos
    xor eax, eax
    mov ax, 0xA0
    mov ds, ax      ; {index: 20 | gdt: 0 | rpl: 00} 20 = 0x14 => Agrego 3 ceros la final
    mov es, ax
    mov gs, ax
    mov ss, ax
    mov ax, 0xB0
    mov fs, ax      ; {index: 22 | gdt: 0 | rpl: 00}
```

Aclaración: Lo que hacemos aquí es mover cr0 a eax para poder setear el bit de modo protegido en 1, y luego moverlo nuevamente a cr0. Luego cambiamos el selector de segmento porque el procesador había preparado las siguientes instrucciones en modo real, la única manera de que las haga en modo protegido, es realizando un jump, para que las ejecute de nuevo. Saltamos al segmento de codigo a la etiqueta de $modoprotegido$. Movemos los índices adecuados a los selectores adecuados(cs, ds, es, gs, ss, fs). También inicializamos la pila en la dirección 0x27000 como lo pide el enunciado.

c y d) Se pide declarar un segmento adicional que describa el area de pantalla en memoria y una rutina para escribir en pantalla

```
[GDT_IDX_VS] = (gdt_entry){
(unsigned short)    0x0F9F,   /* limit[0:15]  */ // 80 x 25 x 2 = 4000 B.
(unsigned short)    0x8000,   /* base[0:15]   */
(unsigned char)     0x0B,     /* base[23:16]  */
(unsigned char)     0x02,     /* type         */ //?
(unsigned char)     0x01,     /* s            */ // En este caso es 'dat'
(unsigned char)     0x00,     /* dpl          */ // level 3 2b = 10?
(unsigned char)     0x01,     /* p            */ //?
(unsigned char)     0x00,     /* limit[16:19] */
(unsigned char)     0x00,     /* avl          */
(unsigned char)     0x00,     /* l            */
(unsigned char)     0x01,     /* db           */ // db = 1 para que sea Kilobyte
(unsigned char)     0x00,     /* g            */ //?
(unsigned char)     0x00,     /* base[31:24]  */
    }
```

Finalmente el segmento de video, aquí lo que haremos es poner la base en 0xb8000 dado que es en esa direccion donde empieza la memoria de video. Luego la memoria de video son 25 filas y 80 columnas de dos bytes cada posición por eso el tamaño es 80*25*2 = 4000 = 0xFA0, por lo tanto el límite es uno menos porque empezamos a contar desde el 0. Como tipo ponemos que es un segmento de datos que se puede escribir y leer y el bit G lo ponemos en 0 porque no lo necesitamos ya que no es mucho lo que tenemos que segmentar. El resto de los campos se mantienen iguales.
Nos quedaría por lo tanto una gdt de la siguiente manera:

Index	Base	Limite	Tipo	S	P	DPL	AVL	L	D/B	G
0x0	0x0	0x0	0x0	0	0	0	0	0	0	0
0x18	0x0	0x6FFFF	0x0A	1	1	0	0	0	1	1
0x19	0x0	0x6FFFF	0x0A	1	1	3	0	0	1	1
0x20	0x0	0x6FFFF	0x02	1	1	0	0	0	1	1
0x21	0x0	0x6FFFF	0x02	1	1	3	0	0	1	1
0x22	0xb8000	0xF9F	0x2	1	1	0	0	0	1	0

```
void screen_ej1(){
//int i;
int j;
for (j = 0; j< VIDEO_COLS; j++){
char_screen temp = {.caracter = '-', .modo = C_FG_WHITE| C_BG_BLACK};
pantalla[0][j] = temp;
}
int i;
for (i = 1; i < VIDEO_FILS - 1; i++){
for (j = 0; j < VIDEO_COLS; j++) {
char_screen temp = {.caracter = '.', .modo = C_FG_BLACK | C_BG_BLACK};
pantalla[i][j] = temp;
}
}

for (j = 0; j< VIDEO_COLS; j++){
char_screen temp = {.caracter = '-', .modo = C_FG_WHITE| C_BG_BLACK};
pantalla[24][j] = temp;
}

}
```

Esta función esta implementada en screen.c

3. Ejercicio 2

En este ejercicio nos encargamos de crear las entradas necesarias de la IDT y de su correspondiente implementacion para poder atender las excepciones que genera el procesador.
Lo primero que hicimos fue completar las entradas de la IDT en el archivo idt.c.

```
//codigo de idt.c
//SUPERVISOR, DPL = 0
#define IDT_ENTRY(numero)
idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);
idt[numero].segsel = (unsigned short) 0x90;
idt[numero].attr = (unsigned short) 0x8E00;
idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0

//USER, DPL = 3
#define IDT_ENTRY_USER(numero)
idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);
idt[numero].segsel = (unsigned short) 0x90;
idt[numero].attr = (unsigned short) 0xEE00;
idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0

//se usa las definiciones de arriba para poder completar el metodo idt_inicializar()
//asi que creamos las entradas para los primeros 19 descriptores de interrupción luego el 32 y 33
void idt_inicializar() {
        IDT_ENTRY(0);
        ...
        //(del 0 al 19)
        ...
        IDT_ENTRY(19);
```

```
        IDT_ENTRY(32);
        IDT_ENTRY(33);
        IDT_ENTRY_USER(0x50);
        IDT_ENTRY_USER(0x66);
```

Para poder atender excepciones(e interrupciones más adelante) se completó el archivo $isr.h$ agregando la siguiente declaración de función ”void_isrX()”donde X es el numero de excepcion declarada para su siguiente implementacion. La implementacion de las mismas se llevo acabo en el archivo $isr.asm$. Y se probo imprimiendo en pantalla el numero de excepción

4. Ejercicio 3

En este ejercicio empezaremos a trabajar con paginación. Lo que haremos es inicializar el directorio de kernel de manera que mapee con identity mapping las direcciones 0x00000000 a 0x0077FFFF. Esto lo haremos con la funcion mmu_inicializar_dir_kernel() . Luego activaremos paginación y vamos a probar que la paginación funciona escribiendo el nombre de grupo en pantalla.

a) Este ejercicio se encarga de pintar de manera genérica la pantalla de manera que quede como indica la figura 9 y 10 del enunciado de este tp. Para este ejercicio creamos 2 funciones en $screen.h$

```
void print_estado_clean(unsigned int addr_buffer){
    ca (*p)[VIDEO_COLS] = (ca (*)[VIDEO_COLS]) addr_buffer;
    int i;
    int j;
    //const char* title = "Titanic Style";
    for (j = 0 ; j < 80 ; j++){
        p[0][j].c = 0;
        p[0][j].a = C_FG_WHITE | C_BG_BLACK;
    }
    //print(title, 0, 0, C_FG_WHITE | C_BG_BLACK);
    for (i = 1 ; i < 24; i++){
        for ( j = 0 ; j < 80 ; j++){
            if (cuadrante_negro(i,j) == 1){
                p[i][j].a = C_FG_WHITE | C_BG_BLACK;
            } else if (cuadrante_estados(i) == 1) {
                if(j == 0 || j == 79){
                    p[i][j].a = C_FG_BLACK | C_BG_BLACK;
                } else if (j > 1) {
                    p[i][j].a = C_FG_BLACK | C_BG_CYAN;
                } else {
                    p[i][j].a = C_FG_BLACK | C_BG_LIGHT_GREY;
                }

            } else{
                p[i][j].a = C_FG_BLACK | C_BG_LIGHT_GREY;
            }
            p[i][j].c = 0;
        }
    }
    //print_numeros_estados();

}
```

Pantalla modo estado: La función se encarga de setear en pantalla los colores de fondo dividiendola en cuadrantes (cuadrante para banderas, para problema y para estados) las constantes C_FG y C_BG están definidas en el archivo $colors.h$ e indican los colores del byte $modo$ dentro de cada pixel de la pantalla de video

```
void print_mapa_clean(unsigned int addr_buffer){
    ca (*p)[VIDEO_COLS] = (ca (*)[VIDEO_COLS]) addr_buffer;
    int i;
    int j;
    for (i = 0; i < 25 ; i++){
        for (j = 0; j < 80 ; j++){
            if (i < 3){
                p[i][j].c = 0;
                p[i][j].a = C_FG_BLACK | C_BG_GREEN;
            } else if ( i == 3 && j < 16){
                p[i][j].c = 0;
                p[i][j].a = C_FG_BLACK | C_BG_GREEN;
            } else if ( i < 24 ) {
                p[i][j].c = 0;
                p[i][j].a = C_FG_BLACK | C_BG_CYAN;
            }
            else {
                p[i][j].c = 0;
                p[i][j].a = C_FG_WHITE | C_BG_BLACK;
            }

        }

    }
}
```

Pantalla modo Mapa: Coloca el color de fondo para esta pantalla

b) Sobre el archivo $mmu.c$ inicializamos el direcctorio de paginas a partir de la dirección 0x27000 recorreremos 1024 entradas (cada entrada son 4 bytes) seteando solo los atributos de lectura y escritura, luego en la primer y segunda entrada van a tener las direcciones 0x28000 y 0x30000 que es donde van a estar las tablas de paginas, Con identity mapping se hace el mapeo desde la posición de memoria 0x00000000 a 0x0077FFFF. Como se ve, se necesitaron 1 direcctorio de páginas y dos tablas. Los únicos atributos que se setean es bit presente en uno y RW en 1 (para las direcciones que se quiere mapear) para todas las demas todo quda en cero

```
void mmu_inicializar_dir_kernel(){
//el direcctorio de página se ubicará en la direccion 0x27000 de la memoria
page_directory_entry* page_directory = (page_directory_entry*) 0x27000;
//Ahora se llenan las entradas del directorio de página
page_directory[0].dir_base = 0x28000 >> 12;
//la primer tabla en la direccion 0x28000
page_directory[0].p = 1;
page_directory[0].rw = 1;
page_directory[0].g = 0;
page_directory[0].ps = 0;
page_directory[0].i = 0;
page_directory[0].a = 0;
page_directory[0].pcd = 0;
```

```
page_directory[0].pwt = 0;
page_directory[0].us = 0;

    //se va a necesitar de 2 tablas para direccionar los 0x77ffff bytes
page_directory[1].dir_base = 0x30000 >> 12;
//la seguna tabla en la direccion 0x30000
page_directory[1].p = 1;
page_directory[1].rw = 1;
page_directory[1].g = 0;
page_directory[1].ps = 0;
page_directory[1].i = 0;
page_directory[1].a = 0;
page_directory[1].pcd = 0;
page_directory[1].pwt = 0;
page_directory[1].us = 0;

int i;
for (i = 2; i < 1024 ; i++){
//todas las demas posiciones del direcctorio va a tener todas las entradas en cero
}
page_table_entry* page_table_0 = (page_table_entry*) 0x28000;
for (i=0 ; i < 1024 ; i++){
  page_table_0[i].dir_base = i;
  page_table_0[i].p = 1;
  page_table_0[i].rw = 1;
  page_table_0[i].us = 0;
  page_table_0[i].pwt = 0;
  page_table_0[i].pcd = 0;
  page_table_0[i].a = 0;
  page_table_0[i].d = 0;
  page_table_0[i].pat = 0;
  page_table_0[i].g = 0;
  page_table_0[i].disponible = 0;
 }
page_table_entry* page_table_1 = (page_table_entry*) 0x30000;
for (i=0 ; i < 896 ; i++){
  page_table_1[i].dir_base = ((i * 0x1000) >> 12) + 0x400;
  page_table_1[i].p = 1;
  page_table_1[i].rw = 1;
  page_table_1[i].us = 0;
  page_table_1[i].pwt = 0;
  page_table_1[i].pcd = 0;
  page_table_1[i].a = 0;
  page_table_1[i].d = 0;
  page_table_1[i].pat = 0;
  page_table_1[i].g = 0;
  page_table_1[i].disponible = 0;
 }
  for (i=896 ; i < 1024 ; i++){
  //se setea todas las demas entradas de la tabla en 0 (cero)
 }

}
```

Aclaración: Hace falta más de ua tabla, ya que una sola tabla puede mapear hasta 0x400000 posiciones de memoria y sin embargo se pide 0x77FFFF, así que hace falta 2 tablas. La primera se usa en su totalidad y la segunda se usarán las 896 primeras entradas. Lo que hacemos para el resto de las entradas es solamente poner que son de lectura/escritura pero que no estan presentes. El procedimiento es igual al que utilizamos para inicializar el directorio. Finalmente en eax devolvemos la dirección donde empieza el directorio de páginas inicializado.

c), d) En esta parte lo que hacemos es setear el bit de paginación en 1 para habilitarla, para esto antes ponemos en cr3 el directorio de página que inicializamos con la función del item b). Luego se escribe el nombre del grupo en pantalla para probar paginación. En el archivo $kernel.asm$

```
; habilitar paginacion
    mov eax, 0x27000
    mov cr3, eax
    mov eax, cr0
    or eax, 0x80000000
    mov cr0, eax
    ; imprimir nombre del grupo
    call print_nombre_grupo
```

donde call_print_nombre_grupo es una función que está en $screen.c$

```
void print_nombre_grupo(unsigned int addr_buffer){
    const char* nombre_grupo = "Nombre Grupo";
    print(nombre_grupo, 1, 0, C_FG_WHITE | C_BG_BLACK, addr_buffer);
}
```

Al finalizar la paginación debe quedar de la siguiente manera:

CR3

Index	PCD	PWT
0x27000	0	0

Directorio de páginas

Index	base tabla	G	PS	A	PCD	PWT	U/S	R/W	P
0	0x28000	0	0	0	0	0	0	1	1
1	0x30000	0	0	0	0	0	0	1	1
..	..	..	..	..	..	..	..	..	..
1023	0x0	0	0	0	0	0	0	1	0

Tabla de páginas 1era pagina

Index	base página	G	PAT	D	A	PCD	PWT	U/S	R/W	P
0	0x0	0	0	0	0	0	0	0	1	1
1	0x1	0	0	0	0	0	0	0	1	1
..	..	..	..	..	..	..	..	..	..	..
..	..	..	..	..	..	..	..	..	..	1
1023	0xFFFFF	0	0	0	0	0	0	0	1	1

Tabla de páginas 2da pagina

Index	base página	G	PAT	D	A	PCD	PWT	U/S	R/W	P
0	0x100000	0	0	0	0	0	0	0	1	1
..	..	..	..	..	..	..	..	..	..	..
895	0x77FFFF	0	0	0	0	0	0	0	1	1
896	0x0	0	0	0	0	0	0	0	1	0
..	..	..	..	..	..	..	..	..	..	0
1023	0x0	0	0	0	0	0	0	0	1	0

5. Ejercicio 4

En este ejercicio se pide implementar la función 'mmu_inicializar_dir_tarea' encargada de inicializar el directorio de páginas y tablas para una tarea, éstas se encontrarán en el área de tierra. Esta al igual que la memoria del kernel, mapea con identity mapping las direcciones 0x000000000 a 0x0077FFFF, y además mapea virtualmente 3 páginas de kb a partir de la dirección 0x40000000. Las dos primeras mapean las dós páginas de cada tarea ubicadas en el área del mar, y la tercera, página del ancla, mapea la primer página (0x00000000). Este método, realiza identity mapping al igual que la función que inicializa la memoria del kernel, y por otro lado copia la memoria de su código correspondiente de la tierra al área del mar, que luego son mapeadas virtualmente con la función 'mmu_mapear_tarea'. En este método, se inicializa un arreglo llamado **paginas_tarea[3*CANT_TAREAS]** que contendrán las páginas físicas donde está mapeado su código en el área del mar. Este se actualizará cada vez que la ubicación de las mismas sea modificada, permitiendo así actualizar su posición en la pantalla. Cabe destacar, que esta función además devolverá la dirección donde fue ubicado el cr3 de la tarea.

a) La función **mmu_inicializar_dir_tarea** se implementó en $mmu.c$ y como dice arriba se encarga de inicializar el directorio de página y la tabla de página para cada tarea. El mapeo de las direcciones desde 0x0000000 hasta 0x0077FFFF es exactamente el mismo que en el ejercicio 3. Copia el código de la tarea (los códigos están ubicados a partir de la posición 0x10000 area de tierra) a su posición indicada en el área del mar (a partir de la posición 0x100000). Para luego hacer el mapeo respectivo de las páginas en las direcciones 0x4000000, 0x4001000 0x4002000 a las páginas recién copiadas al área del mar.

b) Las funciones **mmu_mapear_pagina** y **mmu_unmapear_pagina** se encargan de eso específicamente, la primera recibe el $cr3$, $dirVirtual$ y $dirFisica$ y se encarga del mapeo en ese directorio de páginas. La segunda la desmapea buscando en la tabla de página (de acuerdo al índice). Ante cualquier cambio en la estructura de paginación se usa la función $tlbflush()$.

6. Ejercicio 5

Para completar este ejercicio se agregaron tres entradas en la IDT para asociar las interrupciones de hardware del reloj (32) y del teclado (33), y dos de software '0x50' y '0x66'. Las tres fueron completadas como puertas de interrupción, con la salvedad que para las dos primeras el DPL es 0, y en las interrupciones de software el DPL es 3, ya que este significa el nivel de privilegio que tiene que tener la tarea para poder llamarla, y en este caso queremos que lo hagan las tareas de nivel usuario. En el caso de las interrupciones por hardware, se deberá avisar al "PIC"que la interrupción fue atendida mediante la función **fin_intr_pic1**. Todas las funciones contenidas en esta sección estarán implementadas en el archivo $sched.c$.

6.1. Rutina del Reloj

Esta llama desde assembler ($isr.asm$) a la función de C, **atender_isr32**, que se encarga de resolver la próxima tarea a ejecutar, y también resuelve si se ejecutarán las banderas de las tareas que estén vivas. Para esto se utilizaron varias variables globales. Dos arreglos de enteros, llamados 'navios' y 'banderas', los cuales poseen el selector de segmento de correspondiente a su ubicación, es decir, navios[0] contendrá el selector de la primera tarea y banderas[0] el selector de la bandera de la tarea 1. En caso de que una tarea sea desalojada, en ambos arreglos se escribirá con un '0' en la ubicación correspondiente. Ademas, para intercambiar entre tareas y banderas se utilizaron dos variables, una llamada 'modo' , esta valdrá 0 = modo tarea, y 1 = modo bandera, y otra 'contador_tareas' la cual permite contabilizar la cantidad de tareas que se ejecutan hasta llegar a tres, momento en que se debe cambiar a modo banderas. Esta función también se encargará de desalojar la tarea si luego de la interrupción de reloj, la tarea actual (la que está en TR) pertenece a una bandera, es decir, la ejecución de esa bandera excedió un tick de reloj, por lo que serán desalojadas esa bandera y su tarea correspondiente.

6.2. Rutina de atención del teclado

Esta rutina se encarga de leer la tecla que provocó la interrupción, y pasarsela como parámetro a la función de C **atender_isr33**. Esta se encarga de manejar lo que se visualizará por pantalla. En el caso que las letras apretadas no sean ni la 'm' ni la 'e' no se realizará ninguna acción. Se tendrá una variable global llamada **'modo_pantalla'** que valdrá 0 ó 1 según el modo de pantalla que este presente al momento de la interrupción. En el caso en que el modo de pantalla coincida con la tecla presionada no se realizará nada. En el caso que se esté en **'modo mapa'** y la tecla presionada sea la 'E' (y en el caso opuesto también), se cambiará la variable 'modo_pantalla' y se copiará la memoria desde el buffer de video correspondiente al Estado hacia la ubicación del buffer de pantalla para que sea mostrado en ese momento.

6.3. Rutina de atención 0x50 (SYSCALLS)

Esta rutina se encarga de pasarle los parámetros de las syscalls a la función de C **atender_isr0x50** y luego saltar a la tarea $idle$. Esta se encarga de resolver según los parámetros el servicio que la tarea solicitó, y en el caso que alguno de éstos devuelvan un resultado FALSE serán desalojadas, al igual que si una bandera llama a esta interrupción. Estos servicios son implementados en el archivo $game.c$.

6.4. Rutina de atención 0x66 (impresión en pantalla de la bandera)

Esta se encarga de actualizar el estado de la bandera en el buffer de Estado, moviendo los datos que generó la función bandera. En el caso que una tarea llame a esta interrupción, será desalojada.

7. Ejercicio 6

En este ejercicio son definidas las entradas en la GDT para las tareas, es decir, las TSS. Para eso, tendrán seteados el bit S=0 y Type=1001b. La diferencia entre las mismas será que para las tareas idle, inicial y las banderas (que serán ejecutadas a nivel kernel) tendrán DPL=0, mientras que las restantes tareas correspondientes a los navíos correrán en nivel usuario, tendrán DPL=3. Cabe mencionar, que la TSS correspondiente a la tarea inicial contendrá datos irrelevantes ya que solo se usará para saltar a esa tarea como primer tarea y por única vez. Todas estas serán inicializadas en el archivo $gdt.c$. Para completar el campo de la dirección base de estas tss's, se declararon variables globales en $tss.c$, llamadas **tarea_inicial, tarea_idle, tss_navios[CANT_TAREAS], tss_banderas[CANT_TAREAS]**.

Luego de inicializadas las TSS's en $gdt.c$ con estos datos, se procedió a completar los datos de cada tss en el archivo $tss.c$. Se utiliza un método llamado **tss_inicializar()**, que se ocupa de llamar a los métodos que inicializan las tss's idle, inicla y tareas. Para la tarea idle, que correrá a nivel kernel, se le estableció el mismo cr3 que el kernel, y fueron mapeadas virtualmente sus páginas de código en el área de la tierra. Para las tareas de las banderas, que correrán a nivel kernel, se les seteó segmentos de código y datos de nivel 0. El eip de las mismas estará seteado donde comienza su código en el área del kernel. En este archivo habrá también un método llamado **tss_reinit_banderas()**, encargado de resetear únicamente los valores que fueron modificados de las tss's luego de que sean ejecutadas todas las banderas. Para las tareas, se utilizó segmentos de código y datos de nivel 3. Los métodos para inicializar las tss's de las tareas y banderas reciben como parámetro el cr3 que les será asignado. Por esto, existe otro método llamado **tss_init_tasks()**, encargado de llamar a la función **mmu_inicializar_dir_tarea(int tarea)**, que inicializa la memoria de la tarea, y devuelve el cr3 que le fue asignado. Por cada tarea, es llamada a esta función una vez y el cr3 que devuelve es asignado a la tarea con su bandera correspondiente. Para saltar a la tarea inicial, se modifica directamente el valor del registro TR y luego se realiza un 'jump far' a la tarea idle, esto se hace dentro del archivo $kernel.asm$. El código para hacerlo es el siguiente:

```
; 0x100 = Selector de segmento de la tarea inicial
mov ax, 0x100
ltr ax
sti

jmp 0xF8:0 ; 0xF8 = Selector de segmento tarea idle
```

8. Ejercicio 7

8.1. Scheduler

Para inicializar las estructuras del scheduler, se implementó la función **sched_inicializar()**, dentro del archivo $sched.c$. Se cuenta con las siguientes estructuras:

- int navios[CANT_TAREAS] y int banderas[CANT_TAREAS]: Ambos poseen en la posición i-ésima los selectores de segmento correspondientes a la tarea y bandera i-ésima.
- short modo: valdrá 0 en modo tarea y 1 en modo bandera.
- int contador_tareas: este se inicializará en 0 y se irá incrementando mientras se está en modo tarea, al llegar a 3, se cambiará a modo bandera y se reiniciará a 0 este contador una vez finalizadas la ejecución de todas las banderas.
- index_bandera_actual e index_navio_actual: estas contendrán el valor del índice correspondiente en los arreglos de los navios y banderas que se estén corriendo en ese momento.

Con estas estructuras se manejará el scheduler cuando llegue una interrupción de reloj. En el archivo $isr.asm$, se llamará desde assembler a la función de C **atender_isr32** encargada de devolver el próximo índice de la tarea a la cual se saltará. Esta función se encargará de resolver si el siguiente índice corresponderá a una tarea o a una bandera. Dentro de esta función, primero se corroborará que la tarea actual en el registro TR no corresponda a una bandera, en caso de serlo será desalojada junto con su tarea. En caso contrario, se establece el modo en que está el scheduler y luego se llamarán a los métodos **sched_proximo_indice()** y **sched_proxima_bandera()** según indique la variable 'modo'.

8.2. Impresión de las excepciones del procesador y otras causas de desalojo

Para resolver este tema, luego de producida una excepción, el kernel saltará a la rutina de expceción _isrX, (siendo X el índice de la excepción en la IDT) ubicada en el archivo $isr.asm$. Esta se encarga mediante un macro llamado **pushear_registros_desalojo** guardar en variables globales el estado de los registros cuando se produjo el desalojo. Luego se llama a la función de C **print_estado_desalojo** (dentro de $screen.c$) encargada de mostrar por pantalla la tarea que fue desalojada y la excepción que se produjo y a su vez de llamar al método que desaloja la tarea (al final de la función). Estas excepciones pueden ser producidas tanto como las tareas o las banderas. Se cuenta con una variable global llamada **'mensajes_error'**, definido en $screen.c$, siendo ésta un arreglo con los mensajes de cada excepción en su posición correspondiente y también con los mensajes de error que pueden desalojara una tarea, que una tarea llame a $Int0x50$, una bandera llame a $Int0x66$ ó que una $SYSCALL$ devuelva $FALSE$ como resultado.

9. Conclusiones

Con la realización de este trabajo práctico se pudo poner en práctica varias herramientas que tienen estos tipos de procesadores. Si bien el trabajo representa una aproximación a lo que sería un kernel utilizado en los sistemas operativos de la actualidad, efectivamente se pudo simular el uso de estas herramientas que provee el procesador para hacer algo totalmente distinto fuera de cualquier sistema operativo, algo que sabíamos de concepto pero nunca habíamos visto 'tangiblemente'.