DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Organización del computador II
Segundo Cuatrimestre de 2017

Squanchy

Integrante	LU	Correo electrónico
Arroyo, Luis	913/13	luis.arroyo.90@gmail.com
Humpiri Arenas, Juan Carlos	842/14	jhumpiri1599@gmail.com
García Gómez, Luis Enrique	675/13	garcia_luis_94@hotmail.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) 4576-3300
http://www.exactas.uba.ar

Contents

1 Introducción **2**

2 Desarrollo **3**

2.1 Ejercicio 1 3

2.2 Ejercicio 2 6

2.2.1 Interrupciones 6

2.2.2 Excepciones 6

2.3 Ejercicio 3 8

2.4 Ejercicio 4 10

2.5 Ejercicio 5 13

2.6 Ejercicio 6 14

2.7 Ejercicio 7 17

1 Introducción

El objetivo del trabajo practico es aplicar todos los conceptos de *System programming*. Para esto vamos a resolver el TP en forma gradual, para facilitar el entendimiento de todos temas.

La idea central es programar un pequeño $Kernel$ con los mecanismos de protección y ejecución concurrente de tareas para luego poder ejecutar un juego con hasta 16 tareas concurrentes a nivel usuario.

la mayor parte del código generado por nosotros fue realizado en el lenguaje de programación C para simplificar la revolución y mejorar la claridad de las soluciones implementadas.

Es importante destacar además que a medida que se resolvieron los problemas, las soluciones fueron probadas para verificar su funcionamiento, pero debido a que no son relevantes a la resolución de los problemas en sí, no fueron incluidos en el código entregado.

2 Desarrollo

2.1 Ejercicio 1

Lo primero que hicimos fue modificar la Tabla de Descriptores Globales (GDT), completando 5 entradas más, a partir del índice 8. Creamos 5 segmentos, los cuales van hacer un segmento de código de nivel 0, un segmento de código de nivel 3, un segmento de dato nivel 0, un segmento de dato nivel 3 y un segmento de dato nivel 0 (éste último usado para vıdeo).

A continuación mostramos como llenamos sus respectivos descriptores de segmento:
Segmentos de código de nivel 0 y de nivel 3 según:

```
GTD[Indice_codigo].Limite = 0xF400
GTD[Indice_codigo].Base = 0x0000
GTD[Indice_codigo].Base = 0x00
GTD[Indice_codigo].Type = 0x0A
GTD[Indice_codigo].S = 0x01
GTD[Indice_codigo].Dpl = 0x00  ó 0x01 (0 si es nivel kernel y 1 nivel usuario)
GTD[Indice_codigo].P = 0x01
GTD[Indice_codigo].Limite = 0x01
GTD[Indice_codigo].Avl = 0x00
GTD[Indice_codigo].L = 0x00
GTD[Indice_codigo].D/B = 0x01
GTD[Indice_codigo].G = 0x01
GTD[Indice_codigo].Base = 0x00
```

Segmentos de dato de nivel 0 y de nivel 3 según:

```
GTD[Indice_dato].Limite = 0xF400
GTD[Indice_dato].Base = 0x0000
GTD[Indice_dato].Base = 0x00
GTD[Indice_dato].Type = 0x02
GTD[Indice_dato].S = 0x01
GTD[Indice_dato].Dpl = 0x00  ó 0x01 (0 si es nivel kernel y 1 nivel usuario)
GTD[Indice_dato].P = 0x01
GTD[Indice_dato].Limite = 0x01
GTD[Indice_dato].Avl = 0x00
GTD[Indice_dato].L = 0x00
GTD[Indice_dato].D/B = 0x01
GTD[Indice_dato].G = 0x01
GTD[Indice_dato].Base = 0x00
```

Y por ultimo el segmento de dato nivel 0 el cual se usara para vídeo:

```
GTD[Indice_video].Limite = 0x8000
GTD[Indice_video].Base = 0xB800
GTD[Indice_video].Base = 0x00
GTD[Indice_video].Type = 0x02
GTD[Indice_video].S = 0x01
GTD[Indice_video].Dpl = 0x00
GTD[Indice_video].P = 0x01
GTD[Indice_video].Limite = 0x00
GTD[Indice_video].Avl = 0x00
GTD[Indice_video].L = 0x00
GTD[Indice_video].D/B = 0x01
GTD[Indice_video].G = 0x00
GTD[Indice_video].Base = 0x00
```

Para pasar a modo protegido, primero debemos tener armada la GDT con los segmentos a utilizar, después debemos habilitar la puerta A20, cargamos la GDT y ahí recién activamos el modo protegido, el cual se activa con el siguiente codigo:

```
XOR EAX, EAX
MOV EAX, CR0
OR EAX, 0x1
MOV CR0, EAX
```

Y para saltar a modo protegido debemos hacer un JUMP FAR, con el selector de segmento de código de nivel 0 de la GDT el cual es 0x40 (como se muestra en la figura 1), y el offset debe ser una etiqueta dentro del código, donde ahí empezara el modo protegido.

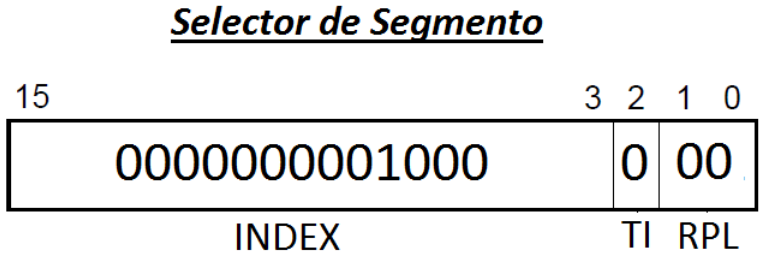

Figure 1: Selector de segmento de código de nivel 0

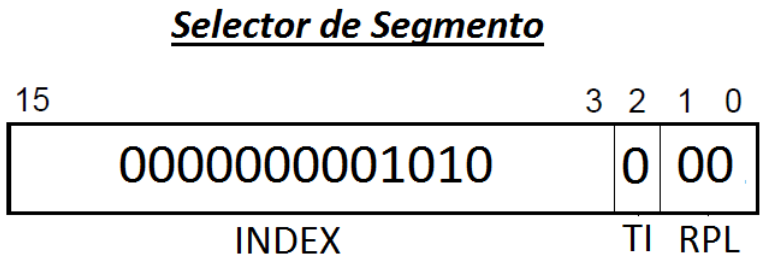

Figure 2: Selector de segmento de dato de nivel 0

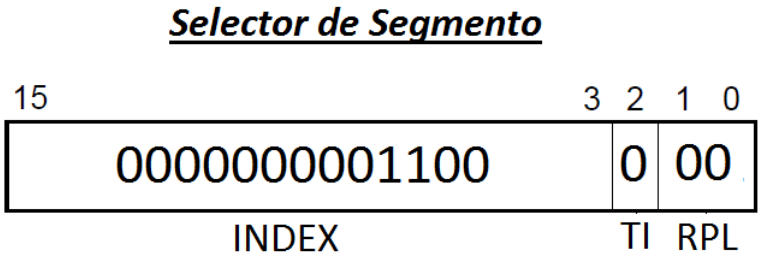

Figure 3: Selector de segmento de vídeo

Y este seria el código:

```
JMP 0x40:MP  ; saltamos al modo protegido
```

Una vez que saltamos a modo protegido, comenzamos a tener la protección por hardware. A partir de aquí es fundamental poner los selectores de segmentos correspondientes a los datos y vídeo habilitándonos efectivamente para el uso de memoria y el frame buffer. Cabe destacar, además, que no necesariamente tenemos una pila que podamos utilizar, por lo que es importante poner los registros correspondientes en

valores que nos sean útiles.

El siguiente código inicia los demás selectores y la pila:

```
mp:
    BITS 32
    ;inicializamos los selectores de segmento
    MOV AX, 0x50 ; AX = Selector de segmento de dato nivel 0
    MOV DS,AX
    MOV ES,AX
    MOV GS,AX
    MOV SS,AX
    MOV AX,0x60
    MOV FS,AX ; AX = Selector de vídeo
    ;establecemos la base de la pila
    MOV EAX, 0x1f400
    MOV EBP, EAX
    MOV ESP, EAX
```

Cabe aclarar que no inicializamos el segmento CS, ya que cuando hicimos el JAMP FAR automáticamente se inicializo con el selector de segmento de código de nivel cero.

2.2 Ejercicio 2

2.2.1 Interrupciones

Para poder atender interrupciones primero debemos llenar la IDT y lo conseguimos llamando a la función `idt_inicializar`.

En ella utilizamos un macro en el cual referimos cada entrada N al segmento de código de kernel, con el offset correspondiente su rutina de atención _isr$N$.

Todas las rutinas de atención de interrupciones deben correr con nivel de privilegio cero, por lo que en sus descriptores usamos el selector de segmento correspondiente al índice 8 de la GDT, el segmento de código de kernel (la figura 4).

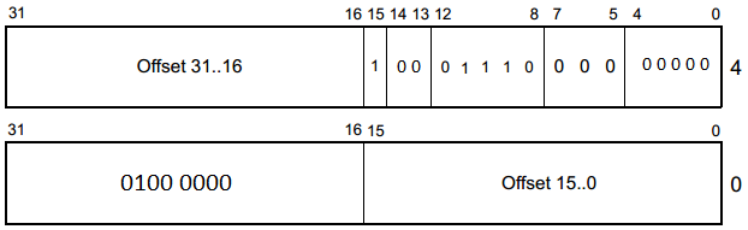

Figure 4: Descriptores interrupt gate

Además, no deben poder ser invocadas por código de usuario, por lo que las definimos con DPL 0 y definimos todas las interrupciones con tipo *interrupt gate*.

2.2.2 Excepciones

En un principio, resolvimos el manejo de excepciones definiendo los primeros veinte índices (de 0 a 19) en la tabla de descriptores de interrupción.

Vector No.	Mnemonic	Description
0	#DE	Divide Error
1	#DB	Debug
2		NMI Interrupt
3	#BP	Breakpoint
4	#OF	Overflow
5	#BR	BOUND Range Exceeded
6	#UD	Invalid Opcode (UnDefined Opcode)
7	#NM	Device Not Available (No Math Coprocessor)
8	#DF	Double Fault
9	#MF	CoProcessor Segment Overrun (reserved)
10	#TS	Invalid TSS
11	#NP	Segment Not Present
12	#SS	Stack Segment Fault
13	#GP	General Protection
14	#PF	Page Fault
15		Reserved
16	#MF	Floating-Point Error (Math Fault)
17	#AC	Alignment Check
18	#MC	Machine Check
19	#XM	SIMD Floating-Point Exception

Figure 5: Protected-Mode Exceptions and Interrupts

Para atender excepciones escribimos rutinas que muestran sus mensajes de error por pantalla, para esto usamos un vector de *char de tamaño 20, así podemos identificar rápidamente por medio del mensaje quien la genero.

Para que el procesador utilice la IDT ejecutamos el siguiente codigo:

```
CALL idt_inicializar
LIDT [IDT_DESC]
; Habilitar interrupciones
STI
```

2.3 Ejercicio 3

Utilizando funciones implementadas en C, y aprovechando las funciones ya provistas por la cátedra, comenzamos limpiando la pantalla.

Después creamos estructuras para el Page Directory Entry (PDE) y Page Table Entry(PTE), las cuales van a contener descriptores con los siguientes atributos (figura 6):

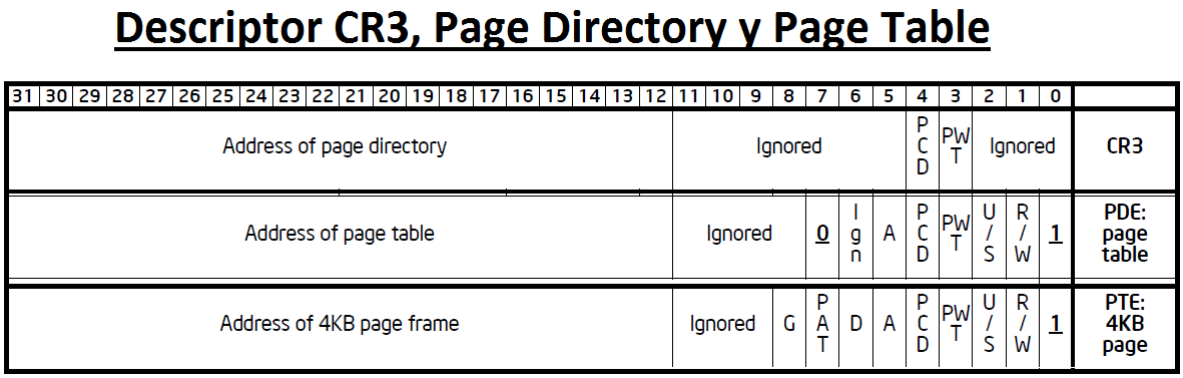

Figure 6: Formato del CR3, Page Directory Entry y Page Table Entry

Una vez de tener las estructuras listas, comenzamos a resolver la función `mmu_inicializar_dir_kernel`, la cual le agregamos la siguientes argumentos:

```
DDB : dirección directorio base.
DTB : dirección tabla base.
```

Donde la dirección de la Page Directory era 0x27000 (como nos decía el enunciado del problema) y la dirección de la Page Table era la 0x28000, ya que PD (Page Directory) tenia 1024 entradas la cual nos consume 4kb de memoria, por eso la PT (Page table) comienza en la dirección 0x28000.

Además los inicializamos con los siguientes atributos:

```
Donde DIRECCIÓN_TABLA_BASE = 0x28000
  PDE[0].p = 1
  PDE[0].rw = 1
  PDE[0].us = 0
  PDE[0].pwt = 0
  PDE[0].pcd = 0
  PDE[0].a = 0
  PDE[0].ign = 0
  PDE[0].ps = 0
  PDE[0].address = (DIRECCIÓN_TABLA_BASE) >> 12

Donde i va 0 a 1023
  PTE[i].p = 1
  PTE[i].rw = 1
  PTE[i].us = 0
  PTE[i].pwt = 0
  PTE[i].pcd = 0
  PTE[i].a = 0
  PTE[i].d = 0
  PTE[i].pat = 0
  PTE[i].g = 0
  PTE[i].ignored = 0
  PTE[i].address = i >> 12
```

Con esto terminamos de inicializar el directorio y tablas de paginas para el kernel usando *identity mapping*, para las direcciones 0x00000000 a 0x003FFFFF lo cual ocupa 4MB de memoria (como se muestra en la figura 7).

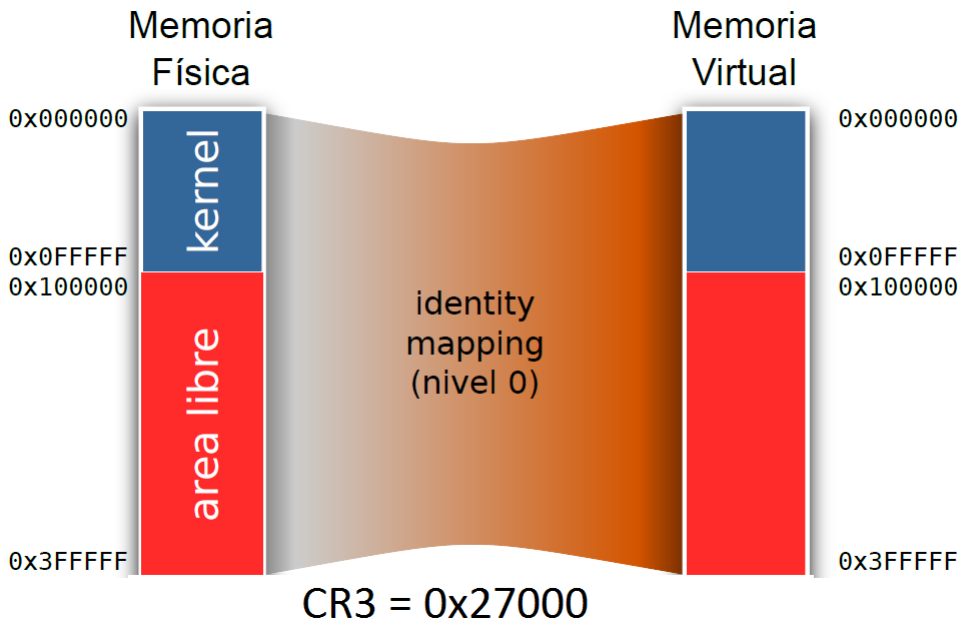

Figure 7: Área de memoria mapeada del kernel con identity mapping

Luego completamos con el siguiente código la activación de paginación:

```
; Inicializar el manejador de memoria
    CALL mmu_inicializar
; Inicializar el directorio de paginas del kernel
    CALL mmu_inicializar_dir_kernel
; Cargar directorio de paginas
    MOV EAX, 0x27000;
    MOV CR3, EAX;
; Habilitar paginación
    MOV EAX, 0x80000000;
    MOV EAX, CR0;
    or ECX, EAX
    MOV CR0, ECX
```

2.4 Ejercicio 4

Para resolver esta parte vamos a usar lo que ya hemos hecho antes, sobre paginación.

Primero vamos a crear una función que nos pueda dar paginas libres para lo que nosotros necesitemos, para esto implementaremos la función `mmu_proxima_pagina_fisica_libre`, la cual va tener una variable global `proxima_pagina_libre`, donde ésta variable la vamos a inicializar con la siguiente función:

```
// Donde INICIO_PAGINAS_LIBRES = 0x100000
void mmu_inicializar() {
  proxima_pagina_libre = INICIO_PAGINAS_LIBRES;
}
```

Luego implementaremos la función `mmu_proxima_pagina_fisica_libre` con el siguiente código:

```
// TAMAÑO_PAGINA = 0x1000
unsigned int mmu_proxima_pagina_fisica_libre() {
  unsigned int pagina_libre = proxima_pagina_libre;
  proxima_pagina_libre += TAMANO_PAGINA;
  return pagina_libre;
}
```

Con esto vamos a poder manejar la memoria área libre del kernel (ver figura 7).

Ahora debemos crear funciones para poder mapear y desmapear una pagina especifica, para lo cual vamos a crear las funciones `mmu_mapear_pagina` y `mmu_unmapear_pagina` las cuales van a tener los siguientes argumentos:

```
Argumentos de la función mmu_mapear_pagina:
    direccion_virtual : dirección virtual.
    CR3 : Aqui nos va llegar la dirección del Page Directory.
    direccion_fisica : dirección física
    atri : los atributos tanto para Page Directory y Page Table

Argumentos de la función mmu_unmapear_pagina:
    direccion_virtual : dirección virtual.
    CR3 : Aqui nos va llegar la dirección del Page Directory.
```

La idea general de la implementación de las funciones son las siguientes:
Para la función `mmu_mapear_pagina`, A partir de sus parámetros, al mapear una página tenemos en cuenta primero si es necesario crear la entrada en el page directory, revisando el bit **P** de la entrada correspondiente a la dirección virtual. De ser así obtenemos una nueva página libre para la tabla de páginas, que inicializamos en 0 para todas sus entradas, y agregamos la entrada en el directorio apuntando a ella usando los atributos de `atri`. Luego de obtener la page table, sea recién creada o previamente existente, asignamos la dirección física alineada a 4KB con los atributos pedidos a la entrada indicada por `virtual`. Hecho esto quedan mapeadas las direcciones `virtual` a `fisica`, y llamamos a la función `tblflush` para limpiar la TLB.
En el contexto de este trabajo solo usamos los bits **P**, **R/W** y **U/S** de los atributos, que coinciden en ambas estructuras.

Para la función `mmu_unmapear_pagina` requerimos una dirección virtual y el directorio en el cual queremos deshacer el mapeo. Con estos datos buscamos la entrada en el directorio. De no estar presente retornamos de la función sin cambios. De lo contrario, obtenemos la entrada correspondiente en la tabla apuntada por ella y anulamos ésta última, seteando el bit **P** en 0. Finalmente, limpiamos la TLB con la función `tblflush`.
Cabe aclarar que la función `tblflush`, es una función implementada por la cátedra.

Una vez que ya podemos mapear una pagina para algún CR3, dirección virtual y dirección física, pasamos a implementar la función `mmu_inicializar_dir_pirata`, la cual va tener los siguientes argumentos:

```
Argumentos de la función mmu_inicializar_dir_pirata:
    fil : posicion de la fila.
    col : posicon de la columna.
    dir_codigo_tarea : dirección del cogido de la tarea.
    J : un puntero a la estructura del jugador.
```

Comenzamos pidiendo 2 paginas libres a la función `mmu_proxima_pagina_fisica_libre`, una para el Page Directory (PD)y otra para Page Table (PT), después llamamos a la función `mmu_inicializar_dir_kernel` para que haga el mapeo de identity mapping los primeros 4MB de memoria. Una vez de haber hecho el identity mapping del kernel, lo que hacemos es mapear la dirección virtual 0x400000 con alguna dirección física que cae adentro del mapa del juego, para poder saber ésta dirección física usamos la función auxiliar `mmu_posXposYfisica`, a la cual le pasamos la posición *col* e fil que nos pasaron como argumentos. La función auxiliar `mmu_posXposYfisica` tiene el siguiente código:

```
//Donde MAPA_BASE_FISICA = 0x500000
uint mmu_posXposYfisica(uint col, uint fil){
return MAPA_BASE_FISICA + (fil * PAGE_SIZE * MAPA_ANCHO) + (col * PAGE_SIZE);
}
```

Una vez obtenido la dirección física que queremos mapear lo único que hacemos es llamar a la función `mmu_mapear_pagina` y le pasamos los siguientes argumentos:

```
Argumentos que le pasamos a la función mmu_mapear_pagina:
    CODIGO_BASE : Es la dirección 0x400000.
    PD : La direccion del Page directory que recién creamos.
    mmu_posXposYfisica(col,fil) : Es la dirección física que queremos mapear en el mapa.
    0x7 : Atributos de la Page Table y Page Directory que son: P = 1, R/W = 1 y U/S = 1.
```

Después de haber terminado el mapeo anterior lo que debemos hacer, es copiar el código de la tarea a la dirección física del mapa según la posición x e y, y ésto lo hacemos con el siguiente codigo:

```
mmu_mapear_pagina(0x500000,rcr3(),mmu_posXposYfisica(col,fil),0x7);
mmu_copiar_pagina(dir_codigo_tarea,0x500000);
mmu_unmapear_pagina(0x500000,pd);
```

Cabe aclarar que a la función `mmu_mapear_pagina` le pasamos como CR3, el actual que se esta ejecutando y eso lo conseguimos con la función `rcr3()`, el cual esta implementado por la cátedra.

Una vez finalizado con la copia del código la tarea, tenemos que resolver el problema de que todas las tareas de un mismo jugador llevan mapeadas todas las paginas del mapa descubierto por los exploradores de ese jugador, la solución que propusimos fue tener dentro de la estructura del jugador un vector de tamaño 4, el cual contendrá 4 page table, las cuales ahí se irán guardando el mapeo de todas las zonas descubiertas por los exploradores, y justamente en la función que estamos implementando debemos usar dicha estructura para encajarselas en su Page directory, así ya las tienen mapeadas. El encaje de las 4 Page Table lo hacemos con el siguiente código:

```
directory_table_entry* page_dir = (directory_table_entry*) PD;
page_dir[2].p        =   1;
page_dir[2].rw       =   1;
page_dir[2].us       =   1;
page_dir[2].address  =   j->paginas[0]>>12;
page_dir[3].p        =   1;
page_dir[3].rw       =   1;
page_dir[3].us       =   1;
page_dir[3].address  =   j->paginas[1]>>12;
page_dir[4].p        =   1;
page_dir[4].rw       =   1;
page_dir[4].us       =   1;
```

```
page_dir[4].address  =  j->paginas[2]>>12;
page_dir[5].p        =  1;
page_dir[5].rw       =  1;
page_dir[5].us       =  1;
page_dir[5].address  =  j->paginas[3]>>12;
```

Y con esto siempre nos aseguramos de que todas las tareas de un mismo jugador, tenga consigo todo el mapeo de lo que ya han descubierto en el mapa.

Por ultimo lo que nos faltaría es mapear las 9 posiciones iniciales, que es donde comienza el pirata en el mapa cuando es lanzado (ver figura 8) y eso lo hacemos usando la función `mmu_mapear_pagina` pasando como argumentos como: dirección virtual le pasamos `mmu_posXposYvirtual(col,fil)` , como dirección física `mmu_posXposYfisica(col,fil)` , como CR3 le pasamos `RCR3()` y los atributos 0x7, para poner los bits P = 1, R/W = 1 y U/S = 1.

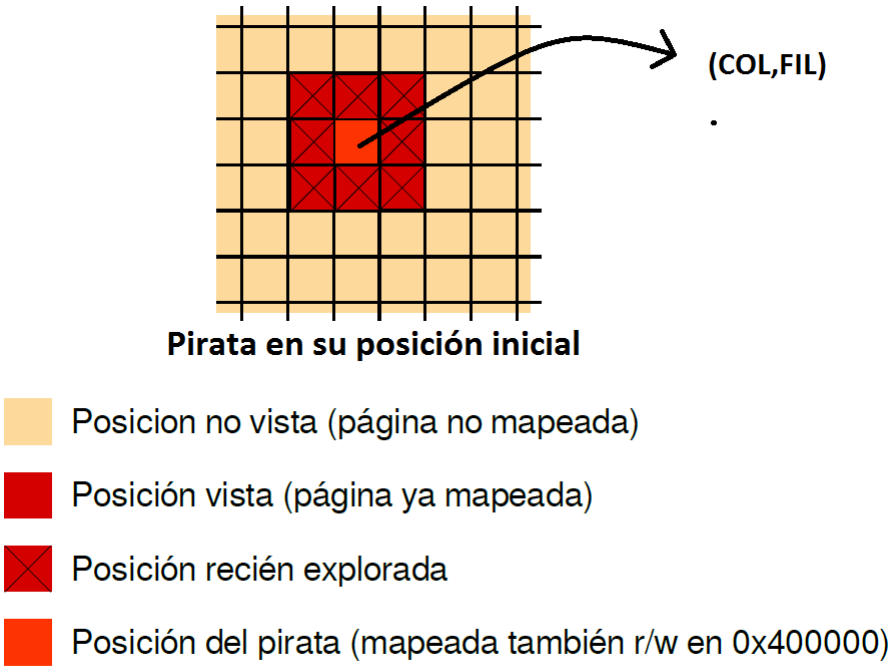

Figure 8: Mapeo del pirata de su posicion inicial y sus adyancetes en el mapa

Cabe aclarar que solo pusimos (col, fil) pero creo que se entiende que col y fil van variar para tener todos sus adyacentes.

2.5 Ejercicio 5

En este ejercicio comenzamos por definir las entradas en la IDT correspondiente a las interrupciones de reloj y de teclado. A éstas las definimos en las posiciones 32, 33 y 70 de la IDT respectivamente, definiéndoles a cada una el offset correspondiente a su rutina de atención de interrupción (definidas posteriormente en isr.asm), el selector de segmento correspondiente al segmento de código de nivel 0 (offset en GDT: 0x40) y los atributos seteados como: bit de presente (P) en 1, la prioridad es 0 y es de 32 bits (bit D en 1) (los atributos quedan expresados como 0x8E00). La interrupción de software 0x46 se diferencia de las anteriores solo en su dpl, el cual es 0x3 para que pueda ser llamada por tareas de nivel usuario, quedando los atributos expresados como 0xEE00.

A continuación debemos escribir la rutina de atención de la interrupción de reloj. Para esto, al comenzar la rutina indicamos que la interrupción fue atendida llamando a la función `fin_intr_pic1` provista por la cátedra. Luego realizamos el llamado a la función `game_tick` y luego terminamos la interrupción. La función `game_tick` realiza un llamado a la función `screen_actualizar_reloj_global` implementada por la cátedra. La funcionalidad total de `game_tick` sera explicada mas adelante.

Para la rutina de atención de la interrupción del teclado, luego de llamar a la función `fin_intr_pic1`, realizamos un código sencillo que imprima la tecla presionada en una posición arbitraria de la pantalla. Esta implementación fue reemplazada por la implementación para agregar una tarea nueva al jugador que oprima la tecla Shift (Right/Left dependiendo que jugador).

A éste ejercicio lo finalizamos definiendo una rutina simple de atención de la interrupción del sistema. Simplemente hacemos que la interrupción, luego de llamar a la función `fin_intr_pic1`, escriba en el registro eax el valor 0x42.

2.6 Ejercicio 6

Definimos en la GDT dos entradas nuevas correspondientes a la TSS inicial (entrada #13) y la TSS idle (entrada #14), ambas con las siguientes características: no están ocupadas (bit Busy en 0), es un descriptor de sistema (bit S en 0), con prioridad de nivel 0 (DPL = 0), tipo = 0x9 , presentes en memoria (P = 1) y con granularidad (G = 0). Las direcciones de memoria y los límites son definidos mediante la función en C `tss_inicializar` y `tss_inicializar_idle`. Además definimos 16 entradas a TSS libres (para lo cual debimos incrementar el tamaño de la GDT) con las características: type = 9, S = 0, dpl = 0, p = 0, g = 0. Para las direcciones bases lo que hicimos fue crear un vector de TSS de tamaño 16 y tomamos como decisión de diseño, fue que los índices pares van a corresponder a TSS de tareas del jugador A y las impares van a corresponder a TSS de tareas del jugador B y los limites son siempre los mismo que son 0x67 para todas las tareas.

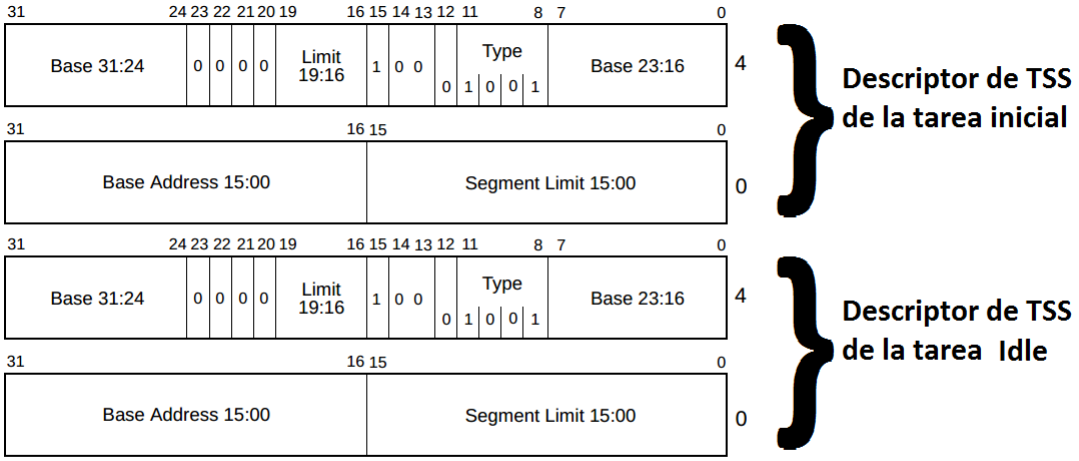

Figure 9: Descriptor de TSS de las tareas inicial e Idle.

Los datos de la entrada de la TSS de la tarea inicial son irrelevantes, motivo por el cual los definimos a todos en cero. A la entrada de la TSS de la tarea IDLE la completamos de la siguiente manera (ver imagen 10).

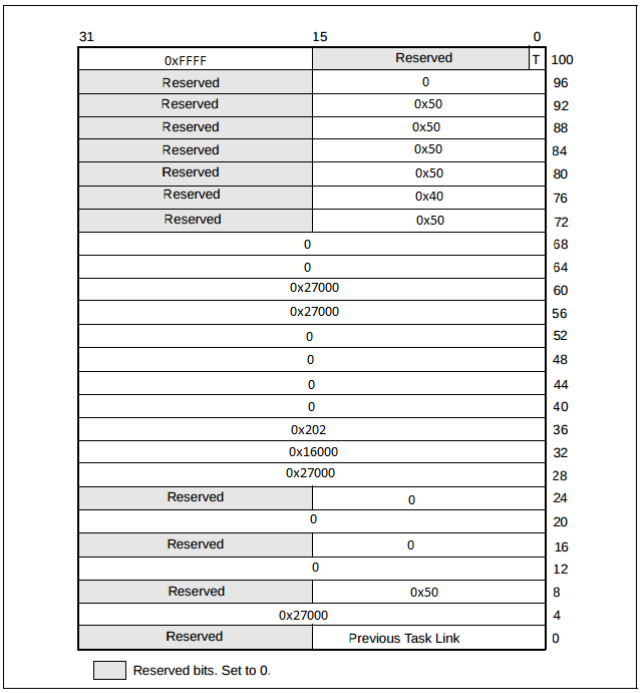

Figure 10: TSS de la tarea idle

La función que completa una TSS libre con los datos pasados por parámetro (el jugador y su posición correspondiente, tipo del pirata, el índice es lugar donde lo guardaremos en la GDT) la declaramos en tss.c y la llamamos `tss_inicializar_pirata`.

La TSS la completamos igual que en la idle a excepción de los siguientes registros:

1. cr3 : mmu_inicializar_dir_pirata
2. eip : 0x400000.
3. ss0 : 0x50. codigo de nivel 0
4. ds : 0x5B. segmento de datos de nivel 3
5. esp0 : nueva pagina de memoria + 0x1000
6. eflags : 0x202. interrupciones activadas.
7. ebp : 0x401000-0xC.
8. esp : 0x401000-0xC.
9. cs, ss, ds, fs, gs : 0x4B. segmento de codigo de nivel 3.

Donde `mmu_inicializar_dir_pirata` le pasamos como parámetro: la posición del pirata, dirección de la tarea (ver imagen 11) y el jugador correspondiente a esta tarea a crear.

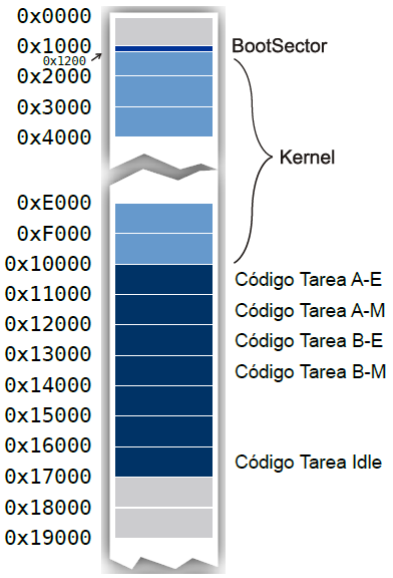

Figure 11: Sector del Kernel donde estas las direcciones del código de las tareas

Luego hicimos fue poner en la pila de la tarea creada, la posición a la cual se va tener que mover (si la tara llega hacer minero) y la eip con valor 0.

El código que necesitamos para saltar a la tarea Idle es el siguiente:

```
; Inicializar tss
call tss_inicializar
; Inicializar tss de la tarea Idle
call tss_inicializar_idle
; Cargar tarea inicial
mov ax, 0x68
ltr ax
; Saltar a la primera tarea: Idle
jmp 0x70:0
```

2.7 Ejercicio 7

Vamos a usar un scheduler construido con arreglo de tamaño 16 que tendrá tareas del jugador A y del jugador B en simultaneo, las tareas pares incluyendo el 0 serán as correspondientes a la del jugador A y las que son impares van a corresponder al jugador B. También vamos a tener un entero que va a ser la tarea actual que se está corriendo (pasaría a ser el índice de la tarea actual) luego, este arreglo va a constar de un bool que es el flag que nos dice si la tarea esta presente o no,un índice que corresponde a la entrada de su TSS en la GDT. Por ultimo tendrá la TSS de la tarea Idle que es la que vamos a usar para iniciar el programa.

La función que vamos a llamar para inicializar el kernel es la función `sched_initialize`

La función `sched_proxima_a_ejecutar` pregunta si el scheduler es vació, si lo es entonces devuelve el selector de segmento de la Idlle. En el caso de que no esté vació, mira el jugador que está ejecutando la tarea, y entonces busca la próxima tarea que corresponda al otro jugador, y en el caso de que el otro jugador no tenga tareas activas, entonces va a buscar alguna próxima de el mismo, en caso de que no hayan mas que una sola tarea, se queda ejecutando en si misma.

Nosotros no implementamos `sched_tick` porque decidimos directamente hacer el llamado a la función `game_tick` dentro de la interrupción de reloj para que pueda ir cambiando los relojes de las tareas en cada task switch.

Hemos modificado la syscall 0x46 manteniendo lo que habíamos hecho en tareas (ejercicio 6) con la diferencia que le agregamos que cada vez que una tarea no cumple los requerimientos necesarios para aplicar la syscall (pasaje de parámetro mapeos, posiciones validas,etc) se procederá a desalojar la tarea quitándolo del scheduler y eliminando su TSS de la GDT y saltamos a la tarea iddle hasta la proxima interrupción de reloj .

El código que agregamos para el intercambio de tareas es el siguiente:

```
_isr32:
  pushad
  ;avisamos al pick que atendemos la interrupion
  call fin_intr_pic1
  ;pedimos el selector de segmento de la proxima tarea a ejecutar
  call sched_proxima_a_ejecutar
  push eax
  ;modificamos el reloj
  call game_tick
  pop eax
  ;pedimos el selector de segmento de la tarea actual
  str ebx
  cmp bx, ax
  ;si son iguales directamente terminamos
  ;si son distinto hacemos el salto a la tarea que nos dio scheduler
   je .fin
  mov [sched_tarea_selector], ax
  jmp far [sched_tarea_offset]
  .fin:
  popad
  iret
```

Modificamos las excepciones del procesador para que además de mostrar el mensaje del tipo de excepción cometida, también desaloje la tarea. Lo hace de la misma forma que en el syscall 0x46 pero saltamos a la tarea idlle para que luego el scheduler haga el cambio de tarea a la siguiente a ejecutar.

Para implementar el mecanismo de debug lo que hicimos fue, establecer una tecla "y" que activa el modo debug, lo que hará es que la primera tarea que cometa una excepción del procesador, mostrará en pantalla todos los registros de propósito general, los registros de control, así como también los estados de la pila de nivel 0 , 3 y los selectores de segmentos (CS, DS, SS, etc). Luego de esto se espera hasta que un jugador apriete la "Y" para quitar el modo debug, restaurando la pantalla a la misma antes de que se cometa la excepción y luego desalojando la tarea como en puntos anteriores.

Para guardar los registros de propósito general,de control y los demás, creamos una pequeña estructura `debug_data_t` en C que con tendrá todos los registros con los mismos nombres que se identifican. Luego desde asembler se la inicializara con los datos correspondientes de la tarea mostrada. Por ultimo, vamos a usar la estructura para mostrar cada campo por pantalla.

EL estado de la pantalla lo conservamos gracias a que creamos una matriz del tipo `ca_s` que contendrá las mismas dimensiones del mapa. entonces, antes de imprimir por pantalla los datos, nos guardamos todo el mapa, tal cual está y luego de que se presione la Y, se cargaran los datos guardados en la pantalla.

-