



**DEPARTAMENTO  
DE COMPUTACION**

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico III

System Programming

Organización del Computador II  
Segundo Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Matias Nahuel Castro Russo	203/19	castronahuel14@gmail.com
Juan Manuel Torsello	248/19	juantorsello@gmail.com
Maximo Yazlle	310/19	myazlle99@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

<b>1. Ejercicio 1</b>	<b>3</b>
1.1. Descriptores de segmentos . . . . .	3
1.2. Pasar a modo protegido . . . . .	3
1.3. Declarar segmento de pantalla . . . . .	3
<b>2. Ejercicio 2</b>	<b>5</b>
2.1. Completar IDT y las ISR . . . . .	5
2.2. Cargar la IDT . . . . .	5
<b>3. Ejercicio 3</b>	<b>6</b>
3.1. Completar nuevas entradas en la IDT . . . . .	6
3.2. Rutina de atención de interrupción del reloj . . . . .	6
3.3. Rutina de atención de interrupción del teclado . . . . .	6
3.4. Interrupciones 88, 89, 100 y 123 . . . . .	7
<b>4. Ejercicio 4</b>	<b>8</b>
4.1. El esquema de paginación . . . . .	8
4.2. Activar paginación . . . . .	9
4.3. Números de libreta . . . . .	9
<b>5. Ejercicio 5</b>	<b>10</b>
5.1. Inicializar estructuras de memoria . . . . .	10
5.2. Rutina de mapeo y desmapeo . . . . .	10
5.2.1. mmu_map_page . . . . .	10
5.2.2. mmu_unmap_page . . . . .	11
5.3. Inicializar directorio y tablas de páginas para una tarea . . . . .	11
<b>6. Ejercicio 6</b>	<b>12</b>
6.1. Definir entradas GDT . . . . .	12
6.2. Completar TSS de la tarea idle . . . . .	13
6.3. Salto a la tarea IDLE . . . . .	13
6.4. TSS tareas Rick y Morty . . . . .	14
<b>7. Ejercicio 7</b>	<b>16</b>
7.1. Inicializar las estructuras de datos del scheduler . . . . .	16
7.2. sched_next_task() . . . . .	16
7.3. Intercambio de tareas por cada ciclo de reloj . . . . .	17
7.4. Rutina interrupciones de software . . . . .	18
7.5. Modo debug . . . . .	18
<b>8. Ejercicio 8</b>	<b>20</b>
8.1. Inicializar pantalla de juego, distribuyendo Mega Semillas . . . . .	20
8.2. Syscall Create Meeseek . . . . .	20
8.3. Syscall Move . . . . .	23
8.4. Syscall Look . . . . .	24
8.5. Syscall PortalGun . . . . .	25
8.6. Finalización de juego . . . . .	26

## 1. Ejercicio 1

### 1.1. Descriptores de segmentos

Lo primero a realizar será completar la Tabla de Descriptores Globales con descriptores de segmento acordes a los segmentos que necesitaremos. En este caso utilizaremos cuatro descriptores: uno de código de nivel de protección 0, datos de nivel de protección 0, código de nivel de protección 3 y datos de nivel 3 que serán las entradas 10 (0xA), 11 (0xB), 12 (0xC) y 13 (0xD) respectivamente.

Como el modelo de segmentación es flat y los segmentos tienen un tamaño de 201 MB, todos los descriptores tienen su base en 0x0 y su límite será 0xC8FF. Este límite es así porque al estar seteado el flag G, el límite en lugar de ser de bytes es de 4kbytes. Después, el flag P (bit de presente) está prendido para indicar que la entrada es válida. D/B indica que el segmento es de 32 bits y DPL el nivel de privilegio del segmento. El tipo 1010 significa que es código y 0010 datos. Que el flag S esté encendido indica que el segmento es de código o datos, y L está en 0 pues debe estarlo si se utiliza la arquitectura IA-32.

La GDT quedaría así:

Índice	Base	Límite	Tipo	G	P	D/B	DPL	S	L
0x00	0x0	0x0000	0000	0	0	0	00	0	0
0x0A	0x0	0xC8FF	1010	1	1	1	00	1	0
0x0B	0x0	0xC8FF	0010	1	1	1	00	1	0
0x0C	0x0	0xC8FF	1010	1	1	1	11	1	0
0x0D	0x0	0xC8FF	0010	1	1	1	11	1	0

### 1.2. Pasar a modo protegido

Dado que nuestros segmentos ocupan 201MB, necesitaremos habilitar la línea A20 usando las funciones provistas por la cátedra. Luego de cargar la GDT con la instrucción “lgdt [GDT\_DESC]”, siendo “GDT\_DESC” nuestra estructura descriptora de la GDT, realizamos el salto a modo protegido. Este se lleva a cabo seteando en 1 el primer bit del registro de control CR0 y realizando un ‘jump far’, utilizando el segmento de código de nivel 0 y un offset adecuado para que la siguiente instrucción a ejecutar sea válida. En nuestro caso, este salto se expresa con la instrucción “jmp IDX\_CODE\_LEVEL\_0:modo\_protegido”, donde “modo\_protegido” es una etiqueta que referencia la sección de nuestro código a partir de la cual el código escrito no debe considerarse más como de 16 bits, y “IDX\_CODE\_LEVEL\_0” vale 0x50, siendo un selector de segmento que apunta al segmento de código de nivel 0 definido en nuestra GDT.

Una vez en modo protegido, inicializamos todos los segmentos de datos al valor “0x58” seleccionando el índice 11 de nuestra GDT (datos de nivel 0). Antes de imprimir un mensaje de bienvenida celebrando el exitoso salto a este modo, se establece la pila del kernel de nivel 0 en la dirección “0x25000” utilizando las instrucciones “mov ebp, 0x25000” y “mov esp, ebp”.

### 1.3. Declarar segmento de pantalla

Para este ítem, lo que debemos hacer es agregar la entrada adicional en la gdt a la cual le asignamos el número 14, donde esta está definida de la siguiente manera:

Índice	Base	Límite	Tipo	G	P	D/B	DPL	S	L
0x0E	0xB8000	0x1F3F	0010	0	1	1	00	1	0

siendo 1F3F el tamaño en bytes de la pantalla multiplicado por 2. Esto es así, ya que un byte se utiliza para el texto y el otro para el formato (80x50x2). En esta entrada, DPL es 0 porque sólo el kernel puede utilizar esta memoria.

A continuación debemos escribir una rutina que se encargue de limpiar y pintar en pantalla las áreas de los mapas, junto a las barras de los jugadores, usando el segmento descripto arriba. Como todavía nos movemos en memoria segmentada, cargamos “1110000b” en el registro “fs” para utilizarlo como selector del segmento de video, y llamamos a la subrutina “init\_pantalla”.

Esta subrutina primero preserva todos los registros con “pushad”, para recuperarlos posteriormente con “popad” y mantener transparencia. A continuación una pasamos a explicar como implementamos la subrutina “init\_pantalla”:

La idea principal de la función es ir recorriendo el area de memoria con el registro ecx, por mas que se lo incremente de a uno, va saltando de a 2 (un byte para el color de fondo y otro para el color del texto). Como la pantalla tiene un ancho de 80 bytes, lo primero que hay que hacer es pintar con el color BLACK (0x0) la primera fila, o sea los primeros 80 bytes. Una vez hecho esto tenemos que definir lo que sería el sector del tablero que va desde el byte 80 hasta el 3280, esto es porque el tamaño del tablero es de 80x40 bytes. Para esta parte usamos la misma idea que para la linea negra, a medida que aumenta el valor de ecx vamos pintando con GREEN (0x2 << 12) hasta que ecx alcance 3280. Una vez terminada la pantalla solo queda definir los 2 rectangulos donde van a ir marcados los puntajes, para esto elegimos la posicion en el la pantalla que nos parecio la mas similar a la presentada en la consigna, dejando 12 bytes en de margen de ambos lados y 40 entre medio de los 2 bloques.



## 2. Ejercicio 2

### 2.1. Completar IDT y las ISR

Cada vez que se genera una interrupción, el sistema debe poder ejecutar la rutina de atención correspondiente a la misma. Para ello se cuenta con la tabla IDT (Interrupt Descriptor Table), en la que cada entrada tiene un selector de segmento, un offset y unos bits de atributo, que determinan atributos como el nivel de protección y el tipo de entrada.

Llenaremos nuestra tabla con las entradas que corresponden a las rutinas de atención de las excepciones. El procesador con el que estamos trabajando (intel 8086) cuenta con veinte excepciones, por lo que completaremos las primeras veinte entradas de la IDT con los siguientes valores: Para el offset utilizaremos la dirección de memoria del código de la interrupción, como selector de segmento utilizaremos el selector del segmento 0x50 (código de nivel 0) y para los atributos el valor 0x8E00 (tipo Interrupt Gate de 32 bits con DPL 0).

Cabe aclarar que para este trabajo vamos a usar siempre las entradas que son de tipo Interrupt Gate, estas deshabilitan las interrupciones cuando se comienza a atender la excepción.

Hasta esta parte del tp todo lo que se pedia hacer con estas excepciones era imprimir en pantalla cual fue el problema que la produjo y que el programa se cuelgue. Para esto dentro de la interrupcion en codigo ASM se llama a la función imprimir\_excepcion que se encarga de, dado un numero de excepcion imprimir el nombre de la excepcion detallado por le manual de intel, una vez terminado esto se realiza un "jmp \$"lo que produce que se cuelgue el sistema.

### 2.2. Cargar la IDT

Para incluir la IDT en el sistema lo que tenemos que hacer es Llamar desde el kernel a la funcion idt\_init la cual se encarga de definir todas la entradas que vamos a usar en la idt (mostrado en el ítem anterior). Una vez hecho esto se carga la dirección y el tamaño de la IDT en el registro IDTR con la instrucción lidt, todos estos valores son definidos en el archivo idt.c.

### 3. Ejercicio 3

#### 3.1. Completar nuevas entradas en la IDT

Tenemos que extender la IDT con seis entradas adicionales: La interrupción del reloj, la del teclado y las cuatro restantes son la 88, 89, 100 y 123 que las vamos a usar más adelante para que las tareas puedan realizar syscalls.

A diferencia de las interrupciones generadas por las excepciones, estas tres interrupciones son recibidas por el PIC; un microcontrolador que se encarga de administrar las interrupciones por prioridad y enviárselas al procesador. Este tipo de interrupciones se llaman externas y pueden ser activadas o desactivadas con el uso de las instrucciones sti y cli, respectivamente. También es importante mencionar que una vez que el procesador atienda una interrupción derivada por el PIC debe hacerle saber que fue atendida. Esto lo realizamos utilizando la función pic.finish1 provista por la cátedra.

Las entradas de reloj y teclado las asignamos a la posición 32 y 33 respectivamente, y las mismas son definidas como dos entradas mas en la tabla (con los atributos de DPL en 0). Ahora bien, para las interrupciones 88, 89, 100 y 123 tuvimos que definir una nueva función IDT\_ENTRY\_3 la cual define las entradas con DPL en 3, ya que estas son interrupciones de software.

#### 3.2. Rutina de atención de interrupción del reloj

```
_isr32:
    pushad
    call pic_finish1    ;avisar al pic que se recibió la interrupción
    call next_clock     ;imprimir el reloj de sistema
    popad
    iret
```

Esta rutina lo único que hace hasta el momento es avisarle al PIC que se atendió la rutina de la interrupción que fue generada por él, y se llama a la función next\_clock implementada por la cátedra para cargar el reloj en pantalla.

#### 3.3. Rutina de atención de interrupción del teclado

```
_isr33:
    pushad
    in al, 0x60
    push eax
    call printScanCode
    add esp, 4
    call pic_finish1    ;avisar al pic que se recibió la interrupción
    popad
    iret
```

```
void print_scan_code(uint8_t scancode){
    if( 0x1 < scancode && scancode < 0xc ){
        print_hex(scancode-1, 2, 78,0 , 0x3D);
    }
}
```

La rutina del teclado que se encarga de leer las teclas y en caso que sea un numero entre 0 y 9, printea en la esquina superior derecha el scancode del numero que corresponda.

### 3.4. Interrupciones 88, 89, 100 y 123

```
global _isr88
global _isr89
global _isr100
global _isr123

_isr88:
    pushad
    mov eax, 0x58
    popad
    iret

_isr89:
    pushad
    mov eax, 0x59
    popad
    iret

_isr100:
    pushad
    mov eax, 0x64
    popad
    iret

_isr123:
    pushad
    mov eax, 0x7b
    popad
    iret
```

Para esta etapa del TP, la implementación de estas interrupciones consiste únicamente en modificar el valor de `eax` con el valor pedido a través de la instrucción `mov`.

## 4. Ejercicio 4

### 4.1. El esquema de paginación

En nuestro sistema utilizaremos paginación con dos niveles de protección, nivel 0 (sistema) y nivel 3 (usuario). Esto no quita que tengamos que seguir usando segmentación en los casos que sea necesario, como en los segmentos de código, datos y video definidos previamente.

Para el uso de paginación se cuenta con las siguientes tablas que son las encargadas de guardar las traducciones entre memoria virtual y física:

- Page Directory es apuntado por el registro CR3, este cuenta con 1024 entradas que llamamos PDE (Page Directory Entry) en el cual cada uno apunta a una Page Table.

- Page Table : tiene 1024 entradas a las que llamamos PTE (Page Table Entry), las cuales apuntan a la base de la pagina física a la cual se mapea la dirección virtual.

Ambas PDE y PTE tienen además una serie de atributos que sirven para conocer las características de cada pagina que se mapea.

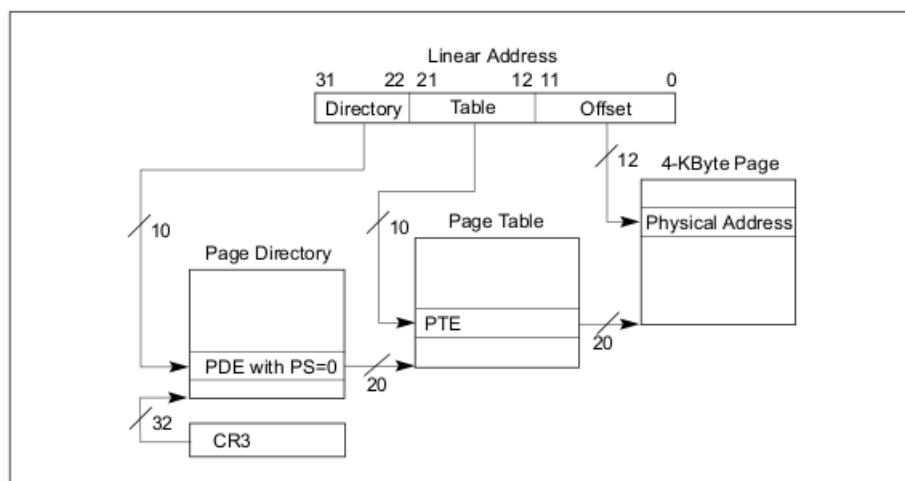


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0			
Address of page directory <sup>1</sup>																				Ignored				P C D	P W T	Ignored				CR3				
Bits 31:22 of address of 4MB page frame										Reserved (must be 0)				Bits 39:32 of address <sup>2</sup>		P A T	Ignored	G	1	D	A	P C D	P W T	U / S	R / W	1	PDE: 4MB page							
Address of page table																				Ignored				0	I g n	A	P C D	P W T	U / S	R / W	1	PDE: page table		
Ignored																										0	PDE: not present							
Address of 4KB page frame																				Ignored				G	P A T	D	A	P C D	P W T	U / S	R / W	1	PTE: 4KB page	
Ignored																										0	PTE: not present							

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging



Para la función **mmu\_init\_kernel\_dir**, encargada de mapear el kernel con identity mapping, lo que hicimos fue crear las `Page_directory` y una `Page_table`, una vez hecho esto, inicializamos cada entrada en 0 para luego poder asignarle los valores que correspondan. Para mapear nuestro kernel de 4MB alcanza con definir una única `page_table`, ya que 4MB son 1024 paginas de 4KB. Una vez que ya tenemos las dos tablas definidas, ponemos correctamente los valores de la primer entrada de la `page_Directory` (la primer entrada es porque las posiciones virtuales hasta la `0x3FFFFFF`, siempre tienen los primeros 10 bits en 0) asignando en 1 el flag de presente y `read_write` para poder acceder a ella, en 0 el flag de `user_supervisor` y poniendo los 12 bits mas significativos de la `page_table` en el atributo `page_table_base`.

Ahora lo único que queda definir son las 1024 entradas de la `page_table`, cada PTE va a tener los mismos atributos que la PDE definida anteriormente, por lo tanto lo único que hay que definir en cada PTE, es la dirección base de cada página, que eso lo hacemos con el mismo índice del ciclo que va recorriendo la `page table`, ya que las paginas que estamos traduciendo son las primeras 1024 de la memoria física.

todos los atributos no mencionados van en 0.

### 4.2. Activar paginación

Después del mapeo debemos activar la paginación para que el procesador la pueda utilizar. Para esto tuvimos que prender el bit 31 del registro de control `CR0`:

```
mov eax, cr0
or  eax, 0x80000000
mov cr0, eax
```

### 4.3. Números de libreta

```
void print_libretas(){
    print(" 203/19, 248/19, 310/19 ", 0, 0, 0x0A);
}
```

Usamos la función `print` provista por la catedra para definir nuestra funcion que imprime en pantalla las libretas.

## 5. Ejercicio 5

### 5.1. Inicializar estructuras de memoria

En la función `mmu_init` nos encargamos de inicializar tanto la primer página libre de kernel como las primeras paginas de las tareas Rick y Morty que vamos a usar más adelante. En el caso del area de kernel, como indica la consigna, la primer pagina disponible del area libre es la `0x100000`. Para las tareas Rick y Morty, en ambos casos la primer página disponible es la `0x8000000`.

### 5.2. Rutina de mapeo y desmapeo

#### 5.2.1. `mmu_map_page`

Lo primero a tener en cuenta en esta función son los parametros que vamos a necesitar para poder hacer el mapeo de una pagina. Nosotros consideramos que con el registro `CR3`, la dirección tanto física como virtual que se quiere mapear y los atributos que va a tener la traducción, tenemos todo lo necesario para hacer el mapeo.

Para arrancar lo que hicimos fue definir los offset con los que vamos a trabajar dentro de cada tabla, el offset del `Page.Directory` se obtiene de los primeros 10 bits de la dirección virtual, y el offset dentro del `Page.Table` se obtiene de los 10 bits siguientes. Para el mapeo no es necesario el offset dentro de la página, ya que si o si se mapea la pagina entera.

```
int off_pd = (virt >> 22);
int off_pt = ((virt << 10) >> 22);
```

Lo próximo es definir los atributos que va a tener la página la cual estamos mapeando, en particular los 3 atributos que se consideran importantes para el tp: el bit de presente (bit 0), el bit de Read/Write (bit 1) y el bit de User/Supervisor (bit 2). En el caso del primero no importa el valor que se pase por parámetro, ya que siempre que se vaya a mapear hay que ponerlo en 1, entonces nos vamos a centrar en los otros 2 bits. Para obtener estos valor alcanza shiftear para izquierda o derecha los valores que se quieran remover.

```
uint8_t FLAG_USER_SUPERVISOR = (uint8_t)((attrs << 30) >> 31);
uint8_t FLAG_READ_WRITE = (uint8_t)((attrs << 29) >> 31);
```

Ahora queda ver como organizar las tablas, para eso primero hay que definir el `Page.Directory` en la posición apuntada por el `CR3`. Una vez que ya podemos acceder al mismo hay que chequear si el PDE ya esta marcado como presente o no. Si ya está presente simplemente definimos la `Page.Table` en la dirección indicada por la tabla, sino la vamos a tener que definir. El hecho que no este presente significa que no tiene una `Page.Table` todavía, por lo tanto la tenemos que crear. Para eso pedimos una página libre del área libre del kernel con la función implementada por nosotros `mmu_next_free_kernel_page()` \*\*. Una vez creada la `Page.Table`, completamos la PDE con los atributos definidos previamente(los que no fueron mencionados van todos en 0) y le asignamos a la PDE la `Page.Table` definida recientemente.

Para terminar nos queda ver la `Page.table` en la entrada que corresponda con el offset de la misma, si esta ya está presente, significa que esa página ya fue mapeada, pero si esta tiene el bit de presente en 0, hay que activarlo y asignarle los atributos de la PTE como en el PDE. El

paso final es asignar al `physical_adress_base` la dirección de la base de la pagina física, o sea la dirección física shifteada 12 bits hacia la derecha.

Se llama a la función `tlbflush()` para que se invalide la cache de traducción de direcciones.

### 5.2.2. `mmu_unmap_page`

Esta función es bastante mas sencilla que la anterior, obteniendo los offset de las tablas de la misma manera, lo único que hay que hacer es acceder a la PTE correspondiente (ambas ya definidas, porque no se tendría que llamar a desmapear una pagina que nunca fue mapeada) y marcar el bit de presente en 0, para indicar que no es una entrada que este activa. La función retorna la pagina física que fue desmapeada.

Se llama a la función `tlbflush()` para que se invalide la cache de traducción de direcciones.

**\*\* funcion `mmu_next_free_kernel_page`**

```
paddr_t mmu_next_free_kernel_page(void) {  
    paddr_t free_page = next_free_kernel_page;  
    next_free_kernel_page += 0x1000;  
    return free_page;  
}
```

Esta función auxiliar se encarga de ir dando las paginas libres del kernel.

## 5.3. Inicializar directorio y tablas de páginas para una tarea


Para esta sección se pide implementar la función `mmu_init_task_dir` que se encarga de, dada la pagina física, la ubicación del código a copiar y la cantidad de paginas, se encargue de inicializar un nuevo `page_directory`, en el cual se van a guardar las traducciones de cada jugador, además esta función tiene que mapear las paginas del kernel y las indicadas en la consigna, copiando el código correspondiente. Para esto lo que hacemos es definir el nuevo `cr3` para las tareas de un jugador (Rick o Morty) con la función `mmu_next_free_kernel_page`. Despues se mapea el kernel y el espacio de la tarea en el nuevo `cr3` utilizando siempre la función `mmu_unmap_page`. Para copiar el código se carga el nuevo `cr3` en el sistema y se copia usando punteros a las posiciones que corresponda. Una vez terminado el copiado del código se carga nuevamente el `cr3` con el que estaba corriendo el sistema y se retorna el nuevo `cr3`, con las paginas pedidas ya mapeadas.

En la consigna se detallan todas las direcciones que entrarían de parámetro, para el caso de Rick y Morty.

## 6. Ejercicio 6

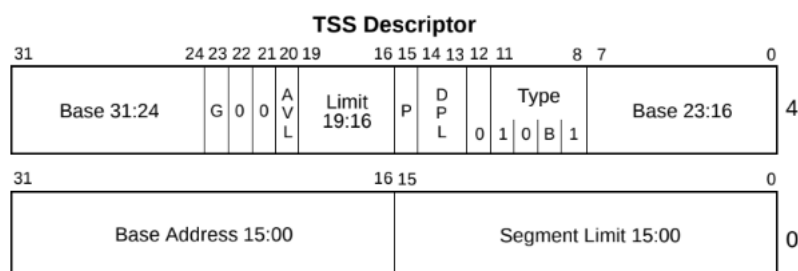
La TSS es un segmento que guarda información sobre la tarea, específicamente su estado o contexto, para que pueda reanudar la ejecución en cualquier momento desde donde se detuvo la misma. Como a los demás segmentos, a cada TSS le corresponde una entrada en la GDT. Cada TSS tiene los siguientes campos:

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
EDI			68
ESI			64
EBP			60
ESP			56
EBX			52
EDX			48
ECX			44
EAX			40
EFLAGS			36
EIP			32
CR3 (PDBR)			28
Reserved	SS2		24
ESP2			20
Reserved	SS1		16
ESP1			12
Reserved	SS0		8
ESP0			4
Reserved	Previous Task Link		0

 Reserved bits. Set to 0.

### 6.1. Definir entradas GDT

Para definir las entradas en la GDT vamos a necesitar definir una nueva TSS(cuyos valores van a ser inicializados más adelante), y conseguir su dirección para ponerla como base en la misma tabla. Los índices los elegimos en las siguientes entradas disponibles (15 para la tarea inicial y 16 para la tarea IDLE). Con estos datos ya listos, llamamos a la función `tss_gdt_entry_init` que se encarga de cargar las entradas de TSS, esta función también toma como parámetro el `dpl` correspondiente a la TSS que se quiera definir, tanto para la tarea inicial como para la IDLE va a valer 0.



La base del segmento va a estar dada por la posición donde este definida la TSS, el limite de los segmentos de TSS va a ser siempre 0x67, ya que son la cantidad de bytes que ocupa una TSS. El tipo va a ser 0x9, porque la tarea no esta siendo atendida, por lo tanto el bit de Busy se setea en 0. El bit de presente y el de avl se setean en 1, el dpl entra por parámetro y en estos dos casos va a ser 0. Cabe aclarar que el bit de sistema está en 0 ya que no es un segmento de código ni de datos y el de granularidad tambien ya que es un segmento menor a 1MB.

```
void init_tss(void) {
    uint32_t base = (uint32_t) &tss_initial;
    tss_gdt_entry_init(IDX_TSS_INIT, base, 0);
}
```

```
void init_idle(){
    uint32_t base_idle_task = (uint32_t) &tss_idle;
    tss_gdt_entry_init(IDX_TSS_IDLE, base_idle_task, 0);
}
```

Hasta el momento asi era la definicion de las funciones encargadas de llamar a inicializar las entradas dentro de la GDT.

## 6.2. Completar TSS de la tarea idle

Para este item lo que hay que hacer es definir los valores importantes que se detallan en el enunciado:

- cr3 = 0x25000 (el mismo que el kernel)
- eip = 0x18000 (la consigna indica que ahí esta el código de la tarea)
- eflags = 0x202 (esta el bit 9 activado, indicando que las interrupciones están habilitadas)
- esp = KERNEL\_STACK (tambien 0x25000, el stack aumenta para las posiciones mas chicas)
- cs = IDX\_CODE\_LVL\_0 (índice de la GDT del segmento de código de nivel 0)
- todos los otros registros de segmento se setean con IDX\_DATO\_LVL\_0 (índice de la GDT del segmento de datos de nivel 0)
- I/OMap se setea en 0xFFFF
- Todos los valores no mencionados previamente se inicializan en 0 para esta tarea.

Esto mismo lo agregamos dentro de la función init\_idle().

## 6.3. Salto a la tarea IDLE

Para poder realizar el intercambio de tareas, tenemos que tener cargado el Task Register (registro que guarda el selector de segmento de la tarea actual), esta es la única razón por la cual existe la tarea inicial. Ahora con el TR cargado correctamente, para saltar a la tarea IDLE hay que hacer un jump far con el selector de segmento de la tarea Idle y el offset en cero, ya que el EIP de la TSS es quien indica donde se arranca a ejecutar la tarea. A continuación mostramos el código que utilizamos para realizar el salto

```
; Cargar tarea inicial
mov ax, IDX_TSS_INICAL
ltr ax

; Saltar a la primera tarea: Idle
jmp IDX_TSS_IDLE:0
```

Donde `IDX.TSS.INICAL` vale `0x78` (15 shifteado 3 para la izquierda) y `IDX.TSS.IDLE` vale `0x80` (16 shifteado 3 para la izquierda).

## 6.4. TSS tareas Rick y Morty

Para este ejercicio implementamos una función que se encarga de crear una TSS para las tareas Rick y Morty principales, y otra que se encargue de los Meeseeks de Rick y Morty. La segunda la vamos a explicar en la sección de la creación de Meeseeks.

La función `tss_creator` toma como parametro al jugador al jugador que se le esta creando la misma. Lo primero que hace la función es ver si estamos creando la TSS de Rick o Morty, en base a eso van a cambiar tres valores : la dirección fisica a la cual se va a mapear la tarea (`0x1D00000` para Rick y `0x1D04000` para Morty), la dirección de memoria donde se encuentra el código (`0x10000` para Rick y `0x14000` para Morty), y la dirección donde esta definida la misma. Para esta ultima creamos dos vectores de TSSs con once posiciones cada uno, la dirección 0 es para la tarea principal y las otras diez para cada Meeseek. Una vez que ya tenemos definidas correctamente esas variables se puede llamar a `mmu_init_task_dir` que se encarga de crear el nuevo directorio de paginas.

Otra de las cosas pedidas por la consigna es pedir una pagina libre del kernel para el stack de nivel 0, eso lo hacemos con la función `mmu_next.free_kernel_page()`. Con esto tenemos todo lo necesario para definir la tss y crear la entrada en la GDT con la función `tss_gdt_entry_init` (para el indice se llama a la función `next.free.tss()` que devuelve el proximo indice libre).

A continuación los campos que se llenan de la TSS:

```
// la pila se setea en el final de la página
tss_new_task->esp0 = stack_level_0 + PAGE_SIZE;

// stack segment de nivel 0, se le asigna el segmento de datos de nivel 0
tss_new_task->ss0 = IDX_DATO_LVL_0;
tss_new_task->cr3 = new_cr3;
tss_new_task->eip = TASK_VIRTUAL_DIR;
tss_new_task->eflags = 0x202;

// la pila se setea en el final de las 4 páginas
tss_new_task->esp = TASK_VIRTUAL_DIR + 4 * PAGE_SIZE;

// como se indica que es un tarea de nivel usuario usa los segmentos de nivel 3
tss_new_task->es = IDX_DATO_LVL_3;
tss_new_task->cs = IDX_CODE_LVL_3;
tss_new_task->ss = IDX_DATO_LVL_3;
tss_new_task->ds = IDX_DATO_LVL_3;
tss_new_task->fs = IDX_DATO_LVL_3;
tss_new_task->gs = IDX_DATO_LVL_3;
tss_new_task->iomap = 0xFFFF;
```

Los campos que no se mencionan se inicializan en 0.

Al final de la función se inicializan los valores del arreglo `info_task`(que contiene dado un índice de la GDT, su información que nos puede ser relevante y útil para futuras implementaciones) por ejemplo, se pone en true el bool `active`, nos dice cual es su índice dentro de la gdt y su número de Meeseek, a que jugador pertenece, y el `flag_loop`(del cual hablaremos luego, pero lo inicializamos en false).

`cr3s` es un arreglo que contiene el `cr3` de las tareas Rick y de las tareas Morty).

También le seteamos a `Sched[player][0]` un puntero a la `info_task` correspondiente. Luego, dado un jugador y su número de tarea (de 0 a 10) nos da información relevante de la tarea que vamos a necesitar al momento de implementar el scheduler).

```
typedef struct info_task{
    bool active;
    uint8_t idx_gdt;
    uint8_t idx_msk;
    uint8_t flag_loop;
    player_t player;
    uint8_t clock;
} info_task_t;
```

## 7. Ejercicio 7

### 7.1. Inicializar las estructuras de datos del scheduler

Inicializamos las variables que van a ir llevando que tarea es la actual y cual fue la anterior ( la vamos a usar para cuando caemos en la tarea IDLE). Creamos una struct

```
typedef struct sched{
    info_task_t* info_task;
} sched_t;
```

para luego declarar

```
sched_t sched[PLAYERS] [11]
```

con la idea de que sched[player] tenga un arreglo con toda información relevante de las tareas del jugador que nos va a ser muy útil luego para poder buscar y devolver la proxima tarea a ejecutar en el siguiente item. A este sched lo inicializamos de manera tal que que todas empiecen estando inactivas y que el flag\_loop arranque apagado(próximamente se explica sobre este flag con más detalle ).

Por ultimo inicializo returning\_debug\_mode en false, el modo debug se explicara luego, pero basicamente lo usamos como flag para saber cuando se ejecuto el modo debug una ejecución anterior y saber cuando restaurar pantalla, etc.

### 7.2. sched\_next\_task()

sched\_next\_tasks es la función que devuelve el índice en la GDT de la próxima tarea a ser ejecutada. Básicamente la idea fue:

- Al querer alternar siempre los turnos de los jugadores, empezamos viendo que jugador corresponde ahora, si es la primera vez que se llama, entonces por criterio nuestro, le setteamos que el turno es de Rick. En todos los demás casos, dado el jugador que ejecuto recientemente, le decimos que nos devuelva el contrario.
- A través de una función auxiliar info\_task\_t\* next(player\_t player), conseguimos la siguiente tarea que tiene que ejecutar el jugador. Esta misma, recorre el arreglo sched[player] y devuelve la primera tarea activa que no tenga flag\_loop en true ("flag"que indica si esa tarea ya se ejecuto en el ciclo de ejecución del jugador). Si al recorrer, todas las tareas activas ya tienen el flag prendido, entonces se los reseteamos a todas sus tareas y busco el primero que satisfaga la condición.
- Si la tarea es de un Meeseek, entonces le actualizamos su reloj llamando a update\_msk\_clocks, que le hace info\_task.clock++ % 4 al info\_task correspondiente y se lo actualiza en el mapa.
- Finalmente actualizamos los índices index, tareaActual y tareaActualAnterior al indice de gdt de la siguiente tarea y lo devolvemos shiteado 3 bits para la izquierda



```

uint16_t sched_next_task(void){
    player_t new_player;
    tareaActual = tareaActualAnterior;
    if(tareaActualAnterior == IDLE){new_player = RICK; }
    else{new_player = info_task[tareaActual].player ? MORTY : RICK;}
    info_task_t* task = next(new_player);
    index = task->idx_gdt;
    tareaActual = index;
    tareaActualAnterior = index;

    bool msk_task = tareaActual > 18 ;
    if (msk_task){
        update_msk_clocks(task);
    }

    return (task->idx_gdt << 3);
}

```

### 7.3. Intercambio de tareas por cada ciclo de reloj

Para realizar el cambio de tareas por cada ciclo de reloj, tenemos que modificar la rutina de interrupción del mismo de manera tal que llame al scheduler cada vez que se ejecuta. Para eso modificamos el código de la siguiente manera:

```

_isr32:
    pushad
    call pic_finish1
    call next_clock

    call sched_next_task
    str cx
    cmp ax, cx           ;Me fijo si la próxima tarea no es la actual
    je .fin
    mov word [sched_task_selector], ax
    jmp far [sched_task_offset]
.fin:
    popad
    iret

```

La función `sched_next_task` como se dijo previamente retorna el índice de la próxima tarea a ejecutar, con `str` se carga en `cx` el valor del `tr` (task-register), para así en la siguiente instrucción compararlo con el valor obtenido por `sched_next_task`, si resulta que el valor de la próxima tarea, es igual al del `tr` (o sea la tarea actual), entonces no se cambia de tarea. Si la próxima tarea no es igual a la actual, (ya teniendo definido el struct de 48 bits `sched_task_offset` `sched_task_selector`) se pone en la parte alta de la struct el valor de `ax`, que es el segmento al cual hay que saltar y finalmente se hace el `jmp far` tomando como parámetro el todo el struct de 48 bits a partir del `offset`.

## 7.4. Rutina interrupciones de software

En esta parte del ejercicio lo único que hay que hacer es modificar el código de las interrupciones 88,89,100 y 123 todas de la misma manera. Lo que hace cada interrupción es llamar a la función `sched_idle` que retorna el índice de la tarea IDLE shifteado 3 y setea la `tareaActual` como la 16 (índice de la IDLE en la GDT). Después se realiza el `jmp far` de igual manera que la explicada en el ítem anterior.

```
_isr88:
    pushad
    mov ebp, esp
    call next_clock ; imprime su reloj
    call sched_idle
    mov word [sched_task_selector], ax
    jmp far [sched_task_offset]
    popad
    iret
```

El código vale tanto para la interrupción 88 como para la 89,100 y 123 solo cambiando el nombre de la rutina.

## 7.5. Modo debug

Lo primero que hay que modificar para el modo debug es el código de las excepciones, ya que si este está activo, se ejecuta cuando ocurre una de ellas. Este modo debe imprimir en pantalla el valor de todos los registros justo en el momento previo a que se produzca la interrupción, para conseguir los mismos, lo que hacemos es apenas entrado en el código de la interrupción pusheamos primero el ESP para tener el valor más cercano al momento del error (después vamos a tener que aumentarlo en 24, ya que se pushean 6 valores a la pila al momento de entrar en la interrupción). Lo próximo es conseguir el EIP, el error code y los eflags, estos 3 valores se pushean en la pila al entrar en la excepción, entonces lo podemos obtener en el código de `asm` buscando en las direcciones apuntadas con el ESP. En el momento justo que se entra a la interrupción, el error code está justo apuntado por el ESP, el EIP está en `[ESP + 4]` y los eflags en `[ESP + 12]`, pero como en la entrada de la función pusheamos el ESP hay que sumarle 4. Después solo queda pushear todos los registros, para luego llamar a la función `imprimir_registros`, que esta imprime todos los valores pasados por parámetros y llama a la función `modo_debug()`, si es que el modo debug está activo, que imprime la pantalla negra y consigue los valores de los registros de control. Es importante antes de imprimir todos los valores en pantalla, hacer un backup de la pantalla actual para al momento de salir del modo debug retomar el juego, este lo hacemos guardándola en **backup\_map**.

La lógica de encendido y apagado del modo debug consistió en agregarle a la interrupción de reloj una condición que cuando se pulsa la letra "Y", se llama a `debug_mode_on_off()` y esta niega el bool indicador de si está prendido o apagado el modo debug. Además si el modo debug está ejecutándose en ese momento (lo vemos con el flag `debug_executing`, que prendemos en la ejecución del modo debug), entonces restauramos la pantalla guardada en **backup\_map** y desactivamos la tarea que llamo a la excepción.

Otro cambio que hicimos fue en `sched_next_task()` para que siempre que el modo debug esté encendido y se esté ejecutando, devuelva que la siguiente tarea es la IDLE y cicle en IDLE hasta que se desactive el modo debug.

También le hicimos otra modificación extra al código de las excepciones, tal que si esta el modo debug encendido no desactive la tarea y lo haga luego `debug_mode_on_off` al momento de detener la ejecución del debug.

No se llegó a implementar el backtrace en el modo debug, por lo que dejamos todo en 0.

## 8. Ejercicio 8

### 8.1. Inicializar pantalla de juego, distribuyendo Mega Semillas

Para la inicialización de la pantalla y la distribución de las semillas usamos la función **game\_init()**, la cual se encarga de imprimir en pantalla todo el interfaz de los marcadores, además en esta función se inicializan las TSS de las tareas principales Rick y Morty con la función explicada en el punto 6 `tss.create`. Con respecto a las semillas, lo primero que hay que tener definido es cuantas semillas va a haber en el juego, para eso tenemos la constante `MAX_CANT_SEMILLAS`, que determina la cantidad inicial de semillas. Para la distribución de las mismas lo que hacemos es ciclar tantas veces como semillas tenga que haber, y en cada ciclo se le asigna una coordenada aleatoria a cada semilla y se la imprime en pantalla, marcando en el arreglo `seed semillas[MAX_CANT_SEMILLAS]` el bit de presente y guardando su coordenada actual.

```
typedef struct seed{
    coordenadas coord;
    bool p;
} seed;
```

Definimos la estructura coordenadas para que el código sea un poco mas claro.

```
typedef struct coordenadas{
    uint8_t x;
    uint8_t y;
} coordenadas;
```

Por ultimo se setean todos los Meeseks como inactivos, en la matriz `meesek_t meeseks[player][10]` poniendo el bit de `p` en 0.

```
typedef struct meesek{
    uint8_t p;                // bit de presente
    uint8_t gdt_index;        // indice en gdt
    coordenadas coord;        // coordenadas actuales
    bool used_portal_gun;     // si usó o no portal gun
} meesek_t;
```

Es importante tener clara esta estructura ya que la vamos a usar mucho de aca en adelante.

### 8.2. Syscall Create Meesek

La rutina encargada de crear los Meeseeks es la `_isr88`, esta principalmente llama a la función `sys_meeseks` que recibe por parámetro la dirección donde está el código a ejecutar, y las coordenadas donde se va a crear.

Lo primero que hace esta función es verificar que el llamado sea correcto, esto significa que la llamada haya sido efectuada por un jugador valido (solamente Rick o Morty), que las coordenadas esten dentro del mapa, y que el jugador no tenga ya activos los 10 Meeseeks (en dicho caso, la función no hace nada y devuelve 0).

Entonces si el Meesek que se esta creando es válido, definimos las coordenadas, para poder chequear si en la posición en cuestion ya había una semilla, para esto se usa la función `coord_in_seed()` que recorre todas las semillas activas verificando si la coordenada del Meesek coincide

con alguna. Si esto pasa se llama a la función `remove_seed()` que se encarga de desactivar la semilla y restar uno `cant_semillas` que lleva la cuenta de las semillas activas. Además se actualiza el marcador de puntos y finaliza la función, devolviendo 0.

En caso contrario, si la coordenada del nuevo Meeseek no cayó en una semilla, entonces creamos al Meeseek. Para esto, pedimos con una función auxiliar, el próximo índice libre de Meeseek (del 0 al 9 inclusive), el cual usaremos en la matriz `meeseeks` mencionada previamente.

Luego toca el momento de crear la TSS del meeseek, para eso llamamos a `tss_meeseeks_creator`, pasándole por parámetro el jugador, el índice del meeseek, la dirección donde está el código a ejecutar que llegó como parámetro, y la coordenada en la cual crear al meeseek. Esta función creará la tarea, actualizando y/o inicializando las variables ligadas al Meeseek y devolverá la dirección virtual en donde fue cargada exitosamente la tarea, que luego de printear en pantalla al Meeseek, `sys_meeseek` devolverá y finalmente la `_isr88` devolverá en EAX dicha dirección.

Una vez terminado la parte de la asimilación de semillas, queda definir la tarea y agregarla en la GDT, para esto creamos una nueva función `tss_create_meeseek` parecida a la función `tss_create`, pero esta toma como parámetro además del jugador el número de Meeseek, la dirección donde arranca el código y las coordenadas del Meeseek.

Esta función `tss_create_meeseek` es parecida a la función explicada anteriormente `tss_create` pero básicamente adaptado a los Meeseeks, con la lógica y reciclaje que necesitan. Dada la coordenada, calculamos la posición en memoria física que le corresponde con `mmu.phy_map_decoder(coordenada)`, y dado el índice de Meeseek y el jugador, nos fijamos el `bool p` que le corresponde en `backup_meeseeks[jugador][indice_meeseek]`, si está en presente quiere decir que ese índice de Meeseek ya existió y vamos a querer reciclarlo. En caso contrario, es la primera vez, por ende no reciclamos. La estructura que tiene cada meeseek para ser reciclado es la siguiente:

```
typedef struct backup_mf{
    bool p;                // presente para reciclar en el futuro
    paddr_t virt;          // dirección virtual a la que corresponde el índice de meeseek
    paddr_t stack_level_0; // dirección del stack level 0
    uint32_t gdt_index;    // índice de gdt
    tss_t *tss;           // puntero al tss
} backup_meeseek;

// matriz para poder acceder dado un jugador e índice de meeseeks a su info de backup
backup_meeseek backup_meeseeks[PLAYERS][MAX_CANT_MEESEKS];
```

Entonces si hay que reciclar reutilizamos la dirección virtual, índice de gdt, y el stack, en caso contrario, las pedimos nuevas con la misma lógica que `tss_create` y los cargamos en la estructura del backup para luego poder ya reutilizarlo.

```
if (reciclar){
    task_virt_address = backup_meeseeks[player][idx_msk].virt;
    mmu_init_task_meeseeks_dir(task_phy_address, code_start, task_virt_address);
    gdt_index = backup_meeseeks[player][idx_msk].gdt_index;
    stack_level_0 = backup_meeseeks[player][idx_msk].stack_level_0;
} else{
    task_virt_address = mmu_next_free_virt_meeseek_page(player);
    mmu_init_task_meeseeks_dir(task_phy_address, code_start, task_virt_address);
    next_free_tss();
    gdt_index = next_free_gdt_idx;
    stack_level_0 = mmu_next_free_kernel_page();

    backup_meeseeks[player][idx_msk].p = true;
    backup_meeseeks[player][idx_msk].virt = task_virt_address;
    backup_meeseeks[player][idx_msk].gdt_index = gdt_index;
    backup_meeseeks[player][idx_msk].stack_level_0 = stack_level_0;
}
```

Después Actualizamos todas las variables y estructuras ligadas al Meeseek:

```
cant_meeseeks[player]++;
info_gdt_meeseeks[gdt_index].idx_msk = idx_msk;
info_gdt_meeseeks[gdt_index].player = player;
info_gdt_meeseeks[gdt_index].ticks_counter = 0;

info_task[gdt_index].player = player;
info_task[gdt_index].active = true;
info_task[gdt_index].flag_loop = 0;
info_task[gdt_index].idx_gdt = gdt_index;
info_task[gdt_index].idx_msk = idx_msk;
info_task[gdt_index].clock = 0;

meeseeks[player][idx_msk].p = 1;
meeseeks[player][idx_msk].gdt_index = gdt_index;
meeseeks[player][idx_msk].coord = coord;
meeseeks[player][idx_msk].used_portal_gun = false;

sched[player][idx_msk + 1].info_task = &info_task[gdt_index];
```

Finalmente cargamos de la tss reciclada o la inicializamos con todos sus valores, en base a si corresponde reciclar o no. Basicamente si hay que reciclar, cargamos todo en la TSS que corresponda, en caso contrario ademas de crearla, la guardamos en su la estructura de backup.

```
backup_meeseeks[player][idx_msk].tss = tss_new_task;
```

Una vez terminada la función por consigna se llama a la tarea IDLE para completar el quantum hasta que se produzca la próxima interrupción de reloj. Para terminar la interrupción se popean registros (que se pushearon en el inicio de la rutina) para que esta sea transparente, luego mueve a EAX el valor retornado por la función sys\_meeseek (0 o la dirección en la que se creo la tarea)y se termina con un iret para retomar la ejecución del código que estaba corriendo antes de la interrupción (por ser iret también se restauran los flags).

### 8.3. Syscall Move

La rutina encargada del movimiento de los Meeseeks es la `_isr123`, esta principalmente llama a la función `sys_move` que recibe por parámetro el desplazamiento en X y en Y.

Al igual que al crear los Meeseeks, al principio hay que chequear que sea una llamada valida,

Esta función va a contar con una serie de restricciones, ya que no cualquier movimiento esta permitido, a continuación se las comenta, la llamada solo puede ser hecha por un Meeseek y no por Rick, Morty o alguna otra tarea y ademas que el movimiento tiene que ser distinto de 0. La última restricción antes de realizar el movimiento es ver que la cantidad de casilleros que se quiere desplazar cumpla con la regla de que inicialmente, la syscall permite moverse a 7 celdas de distancia, contando estas como la suma de los valores absolutos de cada coordenada (distancia Manhattan). Por cada tick de reloj sobre una tarea Mr Meeseeks, su capacidad de moverse se ve degradada a razón de 1 celda por cada dos ticks de reloj, hasta un mínimo de 1 casillero por tick. esto se verifica con la estructura `info_gdt_meeseeks[].ticksCounter` que usa la función `ticks_counter()` que en cada ciclo de reloj modifica una el valor mencionado.

```
uint8_t idx_msk = info_gdt_meeseeks[tareaActual].idx_msk;
player_t player = info_gdt_meeseeks[tareaActual].player;
coordenadas coord_actual = meeseeks[player][idx_msk].coord;
```

De la misma estructura sacamos esa información que nos va a servir para realizar toda la lógica del movimiento.

Como ya verificamos que el movimiento no sea nulo, lo primero que tenemos que hacer es limpiar el casillero donde estaba la M, para esto se llama a la función `clean_cell()` que toma como parametro la coordenada actual y pone el casillero en verde. Como se especifica en la consigna que el mapa es circular, para el calculo de la nueva coordenada decidimos implementar la función `new_position()` que tiene en cuenta los casos donde se sale del mapa. Una vez calculada la nueva posición hay que chequear el caso que haya encontrado una semilla, para esto se llama a la función `coord_in_seed()` explicada en la sección anterior, y en caso de que coincida con una semilla se llama a la función `meeseek_found_seed()`, a continuacion la explicación de la misma:

Esta función toma tres parametros, el jugador que encontro semilla, el indice del Meeseek y el indice de la semilla. En un principio se encarga de marcar el meeseek como inactivo, llama a `remove_seed()` para eliminar la semilla y actualizar las variables correspondientes y se encarga de actualizar el mapa y el marcador de puntos.

El paso siguiente es resetear las siguientes variables (la variable relacionada con el portal gun se va a explicar más adelante):

```
cant_meeseeks[player]--;
info_gdt_meeseeks[tareaActual].ticks_counter = 0;           // el contador de ticks
meeseeks[player][idx_msk].used_portal_gun = false;         // el uso del portalgun
info_task[tareaActual].active = false;                     // se marca la tarea como inactiva
info_task[tareaActual].flag_loop = true;
info_task[tareaActual].clock = 0;
```

Después se enciende el flag que va a permitir la reutilización de la entrada en la GDT, dirección virtual y el stack de nivel 0. Para terminar la función lo único que queda es desmapear la dirección virtual del Meeseek.

Retomando la explicación de la función `sys_move`, ahora solo queda ver el caso en que no haya encontrado semilla. En este caso se actualiza el mapa, y hay que remapear la tarea a la posición que corresponda en el la memoria, cabe aclarar que en la consigna se pide que cada Meeseek este mapeado en memoria en la dirección que corresponda con su ubicación en el tablero. Para esto primero se llama a la función `mmu_phy_map_decoder()` que dada una coordenada calcula la posición en la memoria a donde hay que mapearla.

```
paddr_t mmu_phy_map_decoder(coordenadas coord) {
    paddr_t phy = PHY_INIT_MAP;
    phy = phy + (coord.x * 2 * 4096) + (coord.y * 80 * 2 * 4096);
    return phy;
}
```

Una vez conseguida la dirección física a donde se va a mapear, se llama a la función `mmu_remap_meeseek()` que lo que hace es mapear con identity mapping la dirección física obtenida recientemente para poder copiar el código de la tarea actual, se copia el código, se desmapea el mapeo temporario para el copiado de código y recién al final se mapea la dirección virtual de la tarea con la dirección física que le corresponde. Se retorna 1 en el registro EAX cuando se pudo efectuar correctamente el movimiento solicitado.

Una vez terminada la función por consigna se llama a la tarea IDLE para completar el quantum hasta que se produzca la próxima interrupción de reloj. Para terminar la interrupción se popean registros (que se pushearon en el inicio de la rutina) para que esta sea transparente, luego mueve a EAX el valor retornado por la función `sys_move` y se termina con un `iret` para retomar la ejecución del código que estaba corriendo antes de la interrupción.

## 8.4. Syscall Look

La rutina encargada de buscar semillas es la `_isr100`, esta principalmente llama a la función `sys_look` que recibe por parámetro un flag indicando si tiene que devolver el desplazamiento en X o en Y.

Como siempre lo primero a verificar es que sea un llamado válido, en este caso lo único que hay que chequear es que la tarea que este llamando sea una tarea Meeseek, si esto se cumple se busca la siguiente información sobre la tarea que esta llamando a la interrupción:

```
uint8_t idx_msk = info_gdt_meeseeks[tareaActual].idx_msk;
player_t player = info_gdt_meeseeks[tareaActual].player;
coordenadas coord_actual = meeseeks[player][idx_msk].coord;
```

Para buscar la semilla mas cercana se recorren todas las semillas activas, calculando la distancia Manhattan entre la coordenada actual del Meeseek y la coordenada de cada semilla. Una vez recorridas todas las semillas se guarda el índice de la que este a menor distancia, para después con el arreglo `semillas[]` poder obtener su posición. Para terminar se guarda la diferencia entre la posición en X de la semilla y la posición en X del Meeseek y lo mismo con el eje Y. Dependiendo del flag que este activo se retorna el desplazamiento en X o en Y.

Una vez terminada la función por consigna se llama a la tarea IDLE para completar el quantum hasta que se produzca la próxima interrupción de reloj. Para terminar la interrupción se popean registros (que se pushearon en el inicio de la rutina) para que esta sea transparente,



luego mueve a AL el valor del desplazamiento en X y a BL el valor del desplazamiento en Y obtenidos por la función `sys.look`. Se termina con un `iret` para retomar la ejecución del código que estaba corriendo antes de la interrupción.

## 8.5. Syscall PortalGun

La rutina encargada del uso de arma de portales es la `_isr89`, esta principalmente llama a la función `sys.use_portal_gun` que no recibe ningun parametro ni devuelve nada. Antes de pasar a la implementación de la función, es importante saber de que se trata el portal gun. Cada Meeseek puede usar una única vez esta arma, y lo que hace es mover a una posición aleatoria a un Meeseek enemigo.

Lo primero que hace la función `sys.use_portal_gun` es verificar que la tarea que lo llame sea una tarea Meeseek, si esto se cumple, al igual que las otros servicios definimos algunas variables que nos van a servir:

```
player_t player = info_gdt_meeseeks[tareaActual].player;
uint8_t idx_msk = info_gdt_meeseeks[tareaActual].idx_msk;

// se va a usar para verificar si el Meeseek ya uso el portal gun
bool used_portal_gun = meeseeks[player][idx_msk].used_portal_gun;
player_t opponent = player ? MORTY : RICK;
uint8_t number_opp_msks = cant_meeseeks[opponent];
```

Ahora entonces, si el Meeseek no uso el portal gun previamente y el enemigo tiene algun Meeseek activo, se realiza toda la lógica del servicio. En un principio lo primero que hacemos es actualizar la variable `meeseeks[player][idx_msk].used_portal_gun` para saber que este Meeseek ya uso el arma de portales. Lo proximo es elegir un Meeseek aleatorio que va a ser el que reciba el ataque del arma, para eso realizamos lo siguiente:

```
//vector que va a tener los índices de Meeseeks activos enemigos
uint8_t idxs_msk[number_opp_msks];

// cargo los idxs de los meeseeks activos del oponente
uint8_t j = 0;
for (uint8_t i = 0; i < MAX_CANT_MEESEEEKS; i++){
    if(meeseeks[opponent][i].p){
        idxs_msk[j] = i;
        j++;
    }
}

uint32_t rdm = rand() % number_opp_msks;
uint8_t idx_msk = idxs_msk[rdm];
```

Una vez que ya tenemos cual va a ser el Meeseek a mover, elegimos aleatoriamente a que posición se va a mover y calculamos el desplazamiento, para luego llamar a la función **`move_portal()`** que básicamente hace lo mismo que la función `sys.move`, pero sin ninguna restricción y toma como parametro el jugador oponente. Algo muy importante antes de realizar el `move_portal` es cambiar la `tareaActual` por el numero de tarea del Meeseek que recibe el ataque del arma de portales, y se realiza un **`lcr3(cr3s[opponent])`** para poner el `cr3` del jugador oponente

y realizar los mapeos en el PD correcto. Una vez terminada la función `move_portal`, se restauran tanto la tarea actual como el `cr3` para no generar problemas al salir de la función.

Una vez terminada la función por consigna se llama a la tarea IDLE para completar el quantum hasta que se produzca la próxima interrupción de reloj. Para terminar la interrupción se popean registros (que se pushearon en el inicio de la rutina) para que esta sea transparente. Se termina con un `iret` para retomar la ejecución del código que estaba corriendo antes de la interrupción.

## 8.6. Finalización de juego

En todas las funciones que implican movimiento o creación de meeseeks, además de chequear si obtuvo una semilla, se verifica si esa semilla fue la ultima.

```
if(cant_semillas == 0){
    end_game();
}
```

En caso que sea la ultima semilla, se llama a la función `end_game()` la cual se encarga de poner el centro de la pantalla en negro, y dependiendo de quien sea el jugador que tenga mas puntos, imprime en pantalla "GANADOR RICK", "GANADOR MORTY" o en caso de empate "EMAPTE" además de indicar que se llego al final del juego con el mensaje "FIN DEL JUEGO".

