


Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

# Trabajo Práctico Nro 3
# System Pro1gramming - Batalla Bytal

## Organización del Computador II

Segundo Cuatrimestre de 2013

Grupo: **Frambuesa a la Crema**

| Apellido y Nombre | LU | E-mail |
|---|---|---|
| Ignacio, Truffat | 387/10 | el_truffa@hotmail.com |
| Lasso, Nicolás | 763/10 | lasso.nico@gmail.com |
| Rodríguez, Agustín | 120/10 | agustinrodriguez90@hotmail.com |

# Índice

# 1. Introducción

En el siguiente informe se describen los módulos implementados que constituyen el código del Trabajo Práctico Nro 3 *Batalla Naval*. Cada módulo descripto incluye una breve descripción de las decisiones de diseño tomádas por el grupo con respecto al procesador *Intel* y sus reglas de desarrollo y, de ser necesario, una explicacion de la implementación. Esto incluye en el TP: configuración de la GDT, pasaje a modo protegido, configuración de la IDT, paginación, TSS y la organización del scheduller.

# 2. Desarrollo y Resultados

## 2.1. Ejercicio 1. GDT

### 2.1.1. Global Descriptor Table

Como ya sabemos, el procesador inicia en "modo real", el cual direcciona a 1 MB de memoria y no posee niveles de protección ni privilegios.
Por eso necesitamos que el procesador pase a "modo protegido", para direccionar a más memoria y poder manejar distintos niveles de protección. Nuestro kernel se encargará de hacer esto.

Antes de iniciar en modo protegido, es imprescindible tener bien configurado la Tabla de Descriptores Globales, la cual contiene a los descriptores de segmento, con el fin de definir características de varias áreas de la memoria.
En el enunciado se piden una segmentacion flat, con 4 segmentos que deben direccionar a 1.75 GB: 2 para código de nivel 0 y 3 respectivamente, y 2 para datos, de nivel 0 y 3 también.

La estructura de un descriptor de segmento es la siguiente:

- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

Para definir los segmentos que nos requieren, los items importantes son:

- BASE: este parametro indica el comienzo del segmento. En los 4 casos, este fue 0 ya que se pidió una segmentacion flat.
- P: Present, este parametro indica si el segmento está presente en la memoria. El valor en los 4 segmentos es 0x01 ya que efectivamente estaban presentes.
- DPL: Nivel de privilegios del descriptor. Dado que se piden dos segmentos de código y dos de datos nivel 0 y nivel3, este parametro varía según cual de estos queremos implementar. Nivel 0 implica DPL = 00b y nivel 3 implica DPL = 11b.
- G. Granularity. Este flag indica si el tamaño del descriptor es mayor o menor que 1 Mb. Esto sucede dado que solo se poseen 20 bits para indicar el tamaño del segmento. En particular, si G = 1 entonces el valor de los 20 bits será multiplicado por 4 Kb provocando que con solo 20 bits pueda representar 4Gb de memoria. En nuestro caso queremos un tamaño de 1.75Gb entonces necesitamos G = 1.
- Limit: Tamaño del segmento. Va para los 4 segmentos lo mismo.
  Tenemos que direccionar a 1.75 GB, que son 1792 MB, que equivalen a 1835008 KB.
  Como G vale 0x01, las unidades deben representarse de 4 KB, por eso dividimos por 4.
  $\frac{1835008}{4} = 458752$
  Pero como la memoria empieza desde el 0, debe ser un número menos: 458751
  $458751 = 0x6FFF$

- Type: Indica si es un segmento de código o de datos. Para el segmento de código de nivel 0 ponemos el valor de 0x08, indicando que es "Execute only". para el segmento de código de nivel 3 se usa 0x0A, *Read / Execute*. Mientras que para los 2 de datos ponemos el valor de 0x02, indicando que son de Read/Write.

También se define un segmento que reservado para el área de la pantalla en la memoria. Sabemos que empieza en la dirección base 0x000B8000, con un tamaño de 0x0F9F. Dado que se utilizará como un segmemto de datos, su tipo es de Lectura/ Escritura.

También necesitamos entradas para cada una de las tareas y sus banderas. Es decir, selectores de TSS. Estos serán definidos de forma dinámica y no hardcodeados, basándose en la posición de su respectivo TSS. Básicamente cada una de estas entradas de la GDT para las TSS fue inicada de la siguiente manera:

- BASE: Dirección donde fue definido el comienzo de la TSS para cada respectiva tarea.
- P: Present. Este flag debe ir seteado para todas las TSS.
- DPL: las tareas corren en nivel 3, por lo tanto, el DPL = 3 salvo para las tareas INICIAL e IDLE que deben correr en nivel 0.
- limit: Como mínimo las TSS tienen un tamaño de 104 bytes es decir 0x67. Esto es como mínimo ya que existe la posibilidad de extender el IO Map Base Address.
- type: este es particular. dado que es un tipo de descriptor de segmento, el valor tiene que ser 0x09.
- S: este flag determina si el descriptor se refiere a un segmento de código o datos o si es de sistema. En este caso como los descriptores de TSS son de sistema S = 0.

### 2.1.2. Pasaje a modo protegido

En función de pasar a ejecutar en modo protegido el manual de *Intel*[1] explicita una serie de pasos que se deben seguir para complir con esto.

- Habilitar A20. al realizar esto habilitamos el acceso a direcciones superiores a 1 Mb de memoria.
- Una vez que tenemos configurada la gdt, guardamos su ubicacion en una variable gdt_desc. Para que luego la instrucción lgdt pueda cargar la direccion de comienzo de la GDT.
- Seteamos el flag PE del registro CR0, que indica "Protected Envirnoment".
- Por último para pasar a modo protegido hacemos un jmp al comienzo del segmento de código de nivel 0.
- Una vez ahí acomodamos todos los segmentos apuntando a datos de nivel 0 y seteamos la pila del Kernel en 0x27000 según lo indicado por el enunciado.

[1] Ver Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 System Programming Guide

## 2.2. Ejercicio 2 IDT

### 2.2.1. Interrupt Descriptor Table

A través de la IDT, definimos donde está el código de las interrupciones que manejaremos.
La estructura de una entrada en la IDT está definida en idt.h y en idt.c son iniciadas todas las entradas. Por medio de una macro se cargan las primeras 20 interrupciones del procesador, que van desde la división por 0 hasta la interrupción SIMD.
Luego son completadas todas las entradas restantes de la tabla con entradas de interrupciones inválidas con el propósito de manejar de alguna forma todas las interrupciones posibles. Algunas de estas son definidas nuevamente:

- Interrupción 0x32: Clock.
- Interrupción 0x33: Teclado.
- Interrupción 0x50: Servicios del sistema (syscalls).
- Interrupción 0x66: Handlers de las banderas.

En isr.asm se encuentra el código donde atendemos estas interrupciones. Saliendo de las 4 interrupciones mencionadas arriba (clock, teclado, syscall, bandera), todas las interrupciones seran atendidas de una forma similar (para esto usamos un macro). Se realizan las escrituras pertinentes en pantalla y despues se desalojara la tarea que la causo. Es importante notar que no todas las interrupciones se imprimen igual, pues algunas traen opcode, asi que en pantalla tenemos un array que nos indica cuales instrucciones tienen opcode y cuales no.
La estructura de una entrada de la idt, definida en idt.h, es la siguiente:

- offset_0_15: primeros 16 bits del offset al entry point, que atenderá la interrupción
- segsel: selector de segmento de codigo de nivel 0 la gdt
- attr: atributos de la entrada: Present, DPL, D. Esto varían según si la interrupción es de Reloj o Teclado que llevan DPL = 00b o Servicios o Banderas cuyo DPL = 11b.
- offset_16_31: segundos 16 bits del offset al entry point.

| Indice | Descripcion | P | DPL | D |
|---|---|---|---|---|
| 0...19 | Ins del procesador | 1 | 0 | 1 |
| 32 | Clock | 1 | 0 | 1 |
| 33 | Teclado | 1 | 0 | 1 |
| 80 | Servicios | 1 | 3 | 1 |
| 102 | Banderas | 1 | 3 | 1 |

### 2.2.2. Proceso para activar interrupciones

Para poder activar todas estas interrupciones y sus respectivos handlers se siguen los siguientes pasos:

- Mediante el uso de la instrucción LIDT [IDT_DESC], cargamos el principio del array donde tenemos cargados todas las interrupciones
- Por último se deshabilita, se resetea y se vuelve a habilitar el pic que obtiene las interrupciones.[2]

[2]Las funciones de deshabilitar, habilitar y resetear fueron provistas por la cátedra.

## 2.3. Ejercicio 3. Paginación

### 2.3.1. Kernel, Identity Mapping

Debemos mapear con Identity mapping las direcciones 0x00000000 a 0x0077FFFF. Para esto fueron necesarios:

- 1 Tabla de Directorios de páginas que empieza en la direccion 0x27000.
- 2 Entradas de tabla de directorios que abarcan los 1.75 Gb de memoria.
- 2 tablas de páginas. La primer Page table posee sus 1024 entradas completas direccionando desde 0x00000000 hasta 0x003FFFFF y tiene como base la dirección 0x28000 y la segunda de 0x40000000 a 0x0077FFFF con dirección base en 0x30000.

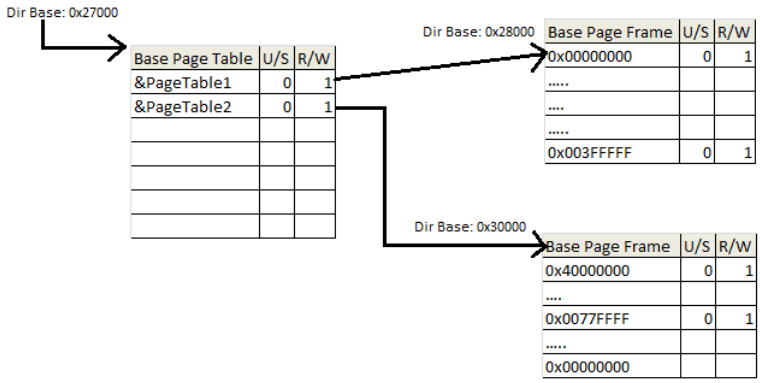
Las entradas de directorio para Kernel son cargadas de la siguiente manera[3]:

- P = 1.
- R/W = 1.
- U/S = 0.
- Direccion de la Page Table = 0x28000 o 0x30000 según corresponda la primer o segunda page table.

Las entradas de Page Table para el Kernel son cargadas de la siguiente manera[3]:

- P = 1.
- R/W = 1.
- U/S = 0.
- Direccion del Page Frame desde 0x00000000 a 0x0077FFFF según corresponda.

A continuación se detalla un esquema para una mejor comprensión de lo explicado:




### 2.3.2. Activación de paginación

Luego de armar el directorio de páginas podemos habilitar la paginación. Para esto seguimo los siguientes pasos:

- Cargar en CR3 la direccion al inicio del directorio de páginas.
- Setear el bit mas significativo del registro CR0.

[3]Se pueden considerar a los flags no declarados como no seteados, es decir, iguales a 0.

## 2.4. Ejercicio 4. Paginación de tareas

### 2.4.1. Paginación de las tareas

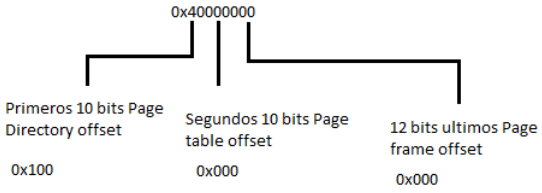
Para la paginación de tareas se necesitaron los siguientes módulos por cada tarea:

- Inicializar un directorio de páginas con 3 entradas, 2 para el Kernel iguales a las descriptas en el ejercicio 3 (es decir, identity mapping) y una para direccionar a las páginas de código y pilas de cada tarea. Este Page directory está definido en la direccion 0x40000000
- Dentro de la Page Table de las tareas se encuentran definidas las entradas de cada página de la tarea. Estas son, 2 entradas para el código de la tarea, 1 para el ancla y (cosa que no fue necesaria pero nos simplifico a la hora de codear) 1 para la pila nivel 0.

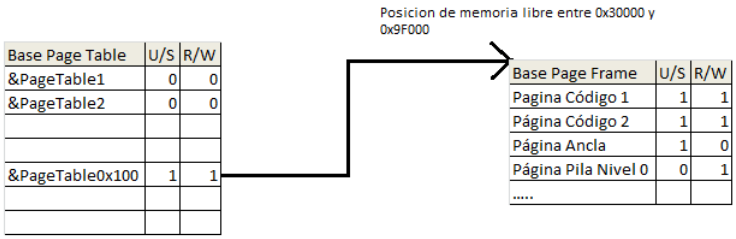
Con el fin de simplificar la cantidad de pasos, las direcciones fisicas de las paginas cada tarea estaran mapeadas en arrays externos. De esta forma no tendremos que buscar dentro de la GDT para buscar una direccion fisica a la hora de hacer otras operaciones, que puede ser costoso en cuestiones de tiempo y dejar un codigo poco claro.

A diferencia del ejercicio anterior, como en este caso estamos mapeando tareas, estas se diferencian de las páginas de Kernel en que los attributos que utilizan son diferentes, es decir, las tareas al correr en nivel 3 requieren que sus Page Directory entries y sus Page table entries posean U/S = 1. De lo contrario no puedo acceder a esas páginas como nos lo demostroó nustra experiencia en la implementación.

El siguiente esquema explica mas simplemente lo nombrado anteriormente. Tener en cuenta que esto debe realizarse por cada tarea y que si bien cada una está mapeada al mismo lugar, la base del Page Directory es distinto para cada tarea, provocando que cada una tenga su propio mapeo. Para empezar, page directory entry que corresponde a la tarea, es la 0x100 ya que 0x40000000 es una direccion virtual entonces la tenemos que decodificar como




Y Luego de esto el esquema de paginación nos quedaría algo así:




| Base Page Table | U/S | R/W |
|---|---|---|
| &PageTable1 | 0 | 0 |
| &PageTable2 | 0 | 0 |
| | | |
| | | |
| &PageTable0x100 | 1 | 1 |
| | | |
| | | |

| Base Page Frame | U/S | R/W |
|---|---|---|
| Pagina Código 1 | 1 | 1 |
| Página Código 2 | 1 | 1 |
| Página Ancla | 1 | 0 |
| Página Pila Nivel 0 | 0 | 1 |
| ..... | | |

Cabe destacar que para poder cumplir con los ejercicios siguientes, una tarea IDLE va a tener que ser mapeada al comienzo de la paginación de lo contrario, no tener una tarea mapeada me imposibilita arrancar el sistema de scheduling que será introducido mas adelante.[4]

[4] Ver sección Ejercicio 7, Scheduling.

## 2.5. Ejercicio 5 IDT / Clocks, Teclados, Syscalls y Banderas

La siguiente sección está dedicada a los *Handlers* o manejadores de las interrupciones. Estos son los códigos que se ejecutan cuando alguna porción del systema produce un error o llama a una interrupción mejor conocida como syscall o servicios del sistema.
Nuestro Kernel cuenta con 4 interrupciones que poseen handlers. Estas fueron mencionadas en el punto 3. Procederemos a explicar cada una de ellas:

- Clock: El clock es una interrupción que se ejecuta cada ciertos ticks de reloj. La misma se encarga de buscar el siguiente selector de segmento según es especificado en la sección 7. y realizar el salto a dicho selector que puede corresponder a una tarea, una bandera o la tarea IDLE.
- Teclado: La interrupción de teclado cumple la función de cambiar el estado de la pantalla entre el mapa si la "m" es presionada y el estado de las banderas y las tareas corriendo si la "e" es presionada.
- Servicios o Syscalls: Esta interrupción brinda al systema una serie de servicios o funciones a las tareas:
  - Fondear: Se accede pasando por parámetro el número 0x923.Esta función permite a la tarea mover por tierra el ancla permitiendole mirar de a una página por vez en tierra. Esta llama a la función implementada en C anclar() ubicada en mmu.c.
  - Cañonear: Se accede pasando por parámetro el número 0x83A, la direccion virtual donde voy a disparar y el buffer de 97 bytes que funciona como misil. Básicamente este servicio nos permite escribir en cualquier lugar del mar un buffer de 97 bytes haciendo que en caso de que en esa dirección se encontrara una tarea enemiga, sus páginas sean corrompidas. Esta llama a la función canionear() implementada en mmu.c
  - Navegar: bajo el número 0xAEF, recibe las nuevas direcciones de las primer y segunda páginas de código de la tarea. Generando que mi tarea se pueda mover por el mar sin ser atrapada por una tarea enemiga. Esta syscall llama a navegar() implementada en C en mmu.c.

  En este caso hemos añadido un handler de error extra que verifica que no sean llamadas por una bandera haciendo nuestro código mas seguro.
- Bandera: Esta interrupción se encarga de imprimir la bandera y dar la impresión de movimiento como si una bandera flameara. Cabe destacar que solo puede ser llamada por una bandera. Si llega a ser llamada por una tarea la misma debe ser desalojada. Esta llama a Bandera() implementada en sched.c

Cabe destacar que las funciones implementadas en C para las syscalls, navegar, canionear y anclar, se encuentran allí dado que manejan páginas de memoria haciendo que ubicarlas en mmu.c sea lo más conveniente para aprovechar todas las funciones y estructuras utilizadas. En el caso de Bandera() se encuentra en sched.c por un razón similar.

## 2.6. Ejercicio 6. TSS

Para que el procesador pueda despachar, ejecutar o suspender múltiples tareas, es necesario salvar el estado de las mismas. La arquitectura provee mecanismos para esto. El segmento de estado (TSS, Task State Segment), es el que se encarga de almacenar la información del estado de una tarea.

Una tarea está identificada por el selector de segmento de su TSS. Y a su vez la TSS es un segmento, por lo tanto debe estar descripto en la GDT junto con los descriptores de segmento de código y datos.[5]

Tenemos 8 tareas y definimos un total de 18 TSS, uno para cada tarea, uno para cada bandera de tarea, uno para la tarea Idle, y otro lo dejamos en blanco para la tarea inicial donde se hace el primer salto.
Las entradas de tss idle y la que tss inicial tienen privilegio de kernel, mientras que las demás están configuradas con privilegios de usuario.

Las TSS se actualizan solas con cada JUMP Far, permitiéndonos así volver más tarde a esa tarea y no perder la información de la misma. Por esto mismo es necesario"incializar" una TSS para que cuando entremos por primera vez la información sea válida. Al momento de inicializar estos segmentos, cada tarea y su flag tendran TSS virtualmente idénticas, con la excepción del eip y pequeños cambios con respecto a las posiciones de las pilas.

Como selectores de segmentos de GDT usamos los que definimos para las tareas (es decir, los de nivel 3), y seteamos el RPL en 0x03 para evitar un GPE.

Una de las grandes ventajas de estar trabajando con direcciones virtuales es que no tenemos que saber la dirección física exacta de + cada tarea para inicializarlas. Sabemos que todas las tareas comparten ciertas direcciones virtuales, asi que seteamos el directorio de página (CR3) correspondiente a esa tarea y podemos usar direcciones idénticas para todas las tareas. Como mencionamos antes, las banderas recibirán datos parecidos, con excepción de las pilas que estarán corridas. (El eip que reciban será indiferente por lo que explicamos más abajo). La pila de nivel 0 sería un caso especial, pero como es acordado en la sección de paginación, la mapeamos en la dirección virtual 0x4000 3000 con el fin de evitar tener que buscarla ahora.
Las TSS de las flags son un caso particular por dos razones. La primera es que el eip no es un valor que sepamos de antemano, sino que depende de cada tarea. Al final de cada tarea hay un offset guardado, que sumándolo a 0x4000 0000 nos da la dirección virtual de la funcion flag. La segunda es que queremos que flag se comporte como una funcion tradicional, es decir, que corra siempre del principio hasta el final (o ser interrumpida). En pocas palabras, no nos interesa la posición donde estuvo la última corrida, sino que nos interesaría volver siempre al comienzo.
Para resolver esto generamos dos funciones que definen el eip del TSS de un flag forma dinámica, y que deben ser corridas antes de saltar a un flag. Por un lado tenemos la función fetch_eip, que se encarga de averiguar el offset de la bandera buscando la tarea en la memoria (recordar que las tareas "naveganz en teoría podrian mutar), y la función reset_eip, que escribe este dato dentro de la TSS.

[5] Ver sección 1 para mas información sobre esto.

## 2.7. Ejercicio 7. Scheduler

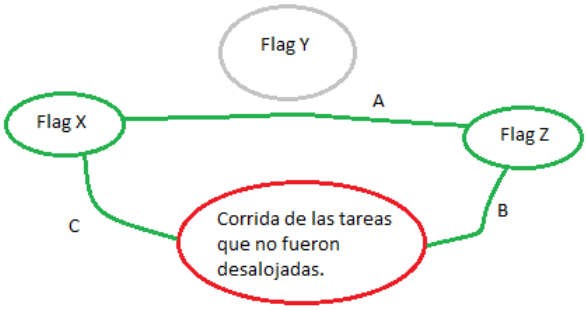
El Scheduler es la estructura más grande y quizás más compleja de nuestro trabajo. Su funcion es simple, coordinar en que orden ocurren los eventos en nuestro Kernel y determinar ciertas acciones como si una bandera excedio el tiempo que le es dado.
Conceptualmente nos imaginamos al Scheduler dividido en dos etapas o dos corridas: una corrida de tareas que es interrumpida por una corrida de banderas. Hay un timer llamado quantum que dictamina cuantos ciclos le queda a la corrida de tareas hasta que sea interrumpido por la corrida de banderas. La corrida de banderas se ejecuta y una vez terminada vuelve a la corrida de tareas con el quantum reiniciado.
Corrida tarea:


- Estado A: Salgo de la ejecución de la tarea X. QUANTUM_RESTANTE = 2 ya que me quedan dos tareas restantes por correr suponiendo que no fueron previamente desalojadas. Si ese fuera el caso vuelve a ejecutar Tarea X hasta agotar los 3 ciclos de QUANTUM. Si TASKS_UP > 0 entonces quiere decir que tengo tareas disponibles, es decir, tengo tareas que aun no han sido desalojadas[6]. Luego evaluo en CONTEXTO en el que me encuentro siendo:
  - EN_IDLE_TOTAL: si solo se puede correr IDLE.
  - EN_IDLE_TAREA: se encuentra ejecutando la tarea IDLE tras haber ejecutado una tarea y dicha tarea haya llamado a una syscall provocando el salto a IDLE. Por lo tanto si QUANTUM_RESTANTE es igual a 0 debería saltar a correr banderas.
  - EN_IDLE_FLAG: se encuentra ejecutando la tarea IDLE tras haber saltado de una interrupción 0x66 luego de haber llamando a la función bandera.
  - EN_TAREA: estoy ejecutando una tarea, con lo cual llegue al clock ejecutando esa tarea.
  - EN_FLAG: estoy ejecutando una bandera y cayó una interrupción de clock provocando que dicha bandera y su correspondiente tareas deban ser desalojadas.

  Dado este CONTEXTO, es que voy a hacer en el siguiente salto a tarea. En nuestra implementación encontraran un Switch() que dependiendo el dicho CONTEXTO será la ejecución que realicemos.
- Estado B: salgo de la ejecución de la tarea Y. Disminuyo el QUANTUM_RESTANTE en 1 y salto a la Tarea Z.
- Estado C: salgo de la ejecución de la tarea Z. Disminuyo el QUANTUM_RESTANTE en 1 dejando el valor igual a 0. entonces el siguiente estado voy a tener que ejecutar una corrida de flags.
- Estado D: salgo de la ejecución de todas las banderas que estaban habilitadas. Si mi tarea X no fue desalojada en la ejecución de su bandera ya sea por un error en su ejecución o porque cayó una interrupción de clock durante su ejecución si haber podido llamar a la interrupción 0x66, entonces puedo saltar a X. El QUANTUM_RESTANTE será restaurado al salir de la corrida de flags con el valor 3. Dado que queremos correr tareas por 3 nuevos ciclos de clock.

[6] Esto no incluye a la tarea IDLE

Con esto explicamos brevemente como está compuesto nuestro scheduler y y su implementación. A continuación hay un breve esquema de la ejecución de las banderas y como sería una corrida suponiendo que ninguna de ellas fue desalojada previamente. En caso de que una de ellas fue desalojada esta será salteada.




- Estado A: se ejecuta la Bandera X correspondientea la tarea X. y se salta a la Bandera Y, pero como la misma fue desalojada previamente, ahora salto directamente a la Bandera Z de la tarea Z.
- Estado B: Luego de la ejecución de la BanderaZ, ya no quedan mas banderas por recorrer dado que la Z es la última y se ejecuta la corrida de los 3 ciclos ejecutando las tareas.
- Estado C: Se acabó nuevamente el QUANTUM_RESTANTE y ejecuto la Bandera X sino fue desalojada durante la ejecución de las tareas.

La interrupción de clock se encarga de realizar todos los saltos y cambios de tareas, exceptuando el salto a idle (que puede ser hecho en cualquier momento). El scheduler es la estructura que le informa hacia donde ir siguiendo. De esta forma, mantenemos el código facilmente segmentado.

Una excepción interesante es el caso en el que no querramos saltar a ningun lado sino seguir en la tarea actual. Por ejemplo, si me queda una sola tarea y estoy en la corrida de tareas aún con quantum me gustaría pertenecer en esa tarea. Para esto el scheduler devuelve el selector de segmento 0, el cual es reconocido por el clock como una instrucción para volver a la tarea anterior (iret) y no realizar ningun salto. (tratar de saltar a una tarea en uso daria error).