DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming

Organización del Computador II

Primer Cuatrimestre de 2016

Grupo: Yo no manejo el raiting, yo manejo un Rolls-Royce

Integrante	LU	Correo electrónico
Luis Enrique Badell Porto	246/13	luisbadell@gmail.com
Nicolas Bukovits	546/14	nicko_buk@hotmail.com
Kevin Frachtenberg	247/14	kevinfra94@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

Resumen

En el siguiente trabajo práctico, se realizó una implementacion de un sistema operativo básico en el cual se aplicaron los conceptos de System Programming vistos en clase. El sistema desarrollado corre en Bochs (programa que permite simular una computadora IBM-PC y realizar tareas de debugging) y permite correr tareas de usuario concurrentemente. La implementación del sistema fue realizada siguiendo una serie de ejercicios en los cuales se desarrolló de forma gradual el sistema completo. La resolucion de los ejercicios y las decisiones tomadas estań explicadas en este informe.

Índice

1. Resolucion de Ejercicios

1.1. Ejercicio 1

a) En este primera parte se empezó a completar la GDT, que contiene los descriptores globales del sistema. El archivo modificado fue el gtd.c que contiene el código de implementación de la GDT. Por normas de Intel, la primera entrada en la GDT tiene que ser nula, por lo que la misma se completó con todos sus valores en 0. Posteriormente se agregaron 4 segmentos requeridos por el enunciado: un segmento para código de nivel 0 (kernel), en el cual se configuró el atributo dpl (privilegio) en 0 para que sólo pueda ser usado por el Kernel, otro segmento para datos también de nivel 0 para el Kernel, un segmento de código de nivel 3 (usuario con dpl 3) y otro segmento de datos de nivel 3. Por requerimiento del enunciado todos estos selectores de segmento fueron configurados para que direccionen los primeros 878 MB de memoria, por lo que los atributos de límite y base son iguales para todos. Las diferencias que presentan son los atributos de dpl antes mencionados y del tipo de selector que fue cargado con 8 para código y 2 para datos.

b) En este punto del ejercicio se implementó el cambio de modo real a protegido y se seteó la pila del kernel en la dirección 0x27000. Este cambio fue realizado en el archivo kernel.asm que contiene el código asm que corresponde al kernel. El código del mismo comienza inhabilitando las interrupciones (mediante la instrucción cli) y cambiando el modo de video a 80 x 50. El código que se agregó para cumplir con el requerimiento en este punto fue habilitar A20, para lo cual sólo fue necesaria llamar mediante a un call a una función externa que ya estaba implementada y se encarga de dicha tarea; y cargar el registro GTDR que contiene la direccción física en donde se encuentra la GDT. Para ello se utilizó la instrucción lgdt a la cual se le pasó como argumento la dirección de la gdt. Posteriormente fue necesario habilitar el modo protegido para lo cual era necesario que el bit de PE (que indica si el modo protegido está habilitado o no) del registro de control CR0 sea seteado en 1. Como no está permitido operar directamente con el registro CR0 se copió su valor al registro de propósito general eax mediante la instrucción MOV y posteriormente se realizó una operación OR del registro eax con el inmediatio de destino en 1, para que el último bit sea seteado en 1. Finalmente se copió el valor actualizado de eax en CR0 y se cambió el modo a protegido realizando un jump far. La instrucción utilizada fue jmp 0x20:mp donde 0x20 representa el selector de código de nivel 0. El cálculo realizado fue el siguiente: índice de código de nivel 0 (que es el 4) multiplicado por 8, es decir shifteado 3 lugares a la izuiqerda. El mp es una etiqueta definida en la próxima instrucción a ejecutar justo inmediatamente despúes del jump, en donde comienza el código que a partir de ese momento se va ejecutar en modo protegido. Las primeras instrucciones que se ejecutan en este punto se encargan de cargar los selectores de segmento que tienen que estar cargados porque es requerimiento del modo protegido. Se cargaron entonces los registros ds, es, gs, fs y ss con el índice de datos de nivel 0 de la GDT (segmentación flat). El cs (code segment no fue necesario cargarlo debido a que la instrucción jmp lo cargó. Se cargaron además los registros esp y ebp de la pila con el valor 0x27000 que es la dirección que indicaba el enunciado en donde está la pila del Kernel (MOV esp, 0x27000 y MOV ebp, 0x27000).

c) Se agregó un segmento adicional en la gdt, que describe el área de la pantalla en memoria que va a ser utilizada solamente por el kernel. El segmento que se agregó por lo tanto fue configurado con el dpl en 0 y es de tipo 2. La GDT entonces pasa a tener en este punto 6 segmentos (teniendo en cuenta al primer segmento nulo).

d) El enunciado solicita que se pinte el área del mapa en un fondo de color. Se implementó una macro en el archivo imprimir.mac denominada pintarPantalla la cual se encarga de pintar el fondo del mapa de color gris. El código de la misma es muy simple ya que se trata de un ciclo que copia a memoria en las posiciones del mapa, en los bytes de atributos de cada pixel, el valor correspondiente al color de fondo de gris. El código de la rutina es el siguiente:

```
%macro pintarPantalla 0

    push ecx
    push edi

    mov ecx , 4000
    xor edi , edi
    %%cicloGris:
        mov word [edi*2 + 0x000B8000] , 0x7700  ;
        inc edi
        loop %%cicloGris
```

```
    pop edi
    pop ecx

%endmacro
```

1.2. Ejercicio 2

a) En esta parte de la implementación se tuvo que completar las entradas en la IDT (que contiene los descriptores de las interrupciones) para asociar diferentes rutinas a las exepciones del procesador. Los archivos modificados fueron el idt.c y el isr.asm. Se implementó la función idt_inicializar() en idt.c la cual se encarga de inicializar las excepciones. Las excepciones configuradas en este punto fueron las contenidas en el rango 0-19 y para configuraralas se definió una macro llamada IDT_ENTRY(numero) cuya implementación es la siguiente:

```
#define IDT_ENTRY(numero)
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);
    idt[numero].segsel = (unsigned short) GDT_OFF_IDX_DESC_CODE0;
    idt[numero].attr = (unsigned short) 0x8E00;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xF
```

La misma se encarga de definir en idt, que es un arreglo de 256 posiciones que contiene las descripciones de las exepciones, en la posicion indicada por numero, el offset, los atributos y el selector de segmento que se trata de un selector de código de nivel 0 ya que van a ser rutinas que va a correr el kernel. Dentro de idt_inicializar() sólo se tuvo que llamar 20 veces a IDT_ENTRY(i) siendo i el número de exepción comprendido en el rango de 0-19. Por otro lado, el archivo irs.asm que contiene el código asm de las rutinas de atención de interrupciones fue modificado agregando una macro _isr: la cual es llamada cuando se produce una exepción y en la misma se usaba el número de exepción que se produjo para mostrar un mensaje en pantalla de la exepción. Para ello se definieron mensajes para cada una de las exepciones. Por ejemplo para el caso de la división por cero se agregó el siguiente mensaje:

```
msj0: db'Divide Error!'
msj0_len equ $ - msj0
```

b) Se solicita que se implemente lo necesario para que se pueda utilizar la IDT y que se pruebe generando una exepción. Para ello en el archivo kernel.asm se realiza un CALL a la función idt_inicializar implementada en el punto anterior y luego se carga el registro IDTR que contiene la dirección física en donde se encuentra alojada la IDT. La instrucción para cargar la misma es lidt a la cual se le pasa como argumento la dirección de la idt. Con estas modificaciones el sistema ya estaba configurado para manejar las exepciones. Para probarlo se agregaron tres instrucciones cuyo propósito era dividir por cero para que se produzca la exepción de Divide Zero. El código agregado fue:

```
xor ebx, ebx
xor eax, eax
div eax
```

El mismo produjo una exepción de dividir por cero, con lo que pudimos probar empíricamente que las exepciones estaban siendo manejadas y funcionaban correctamente. Dichas líneas de código fueron comentadas luego de que se corroboró que funcionaba para que no se produzca la exepción y el kernel siga ejecutando normalmente, y si eventualmente en algún momento se deseaba volver a verificar su funcionalidad sólo se tengan que descomentar. Además nos sirvió como documentación y registro de lo que se fue implementando.

1.3. Ejercicio 3

a) En este punto se pide implementar una rutina que limpie el buffer de video y lo pinte como indica el enunciado. Para esto, se realizó una macro definida en el archivo imprimir.mac cuyo propósito es pintar la pantalla como indica el enunciado. Dentro de esta macro se utiliza la otra macro previamente definida de pintarPantalla.

b) En este punto se tuvo que realizar una de las partes más importantes e interesantes del sistema, en la cual se escribieron las rutinas que inicializan el directorio y las tablas de páginas del kernel. El archivo modificado fue el mmu.c. Se implementó la función mmu_inicializar_dir_kernel cuyo código es el siguiente:

```
void mmu_inicializar_dir_kernel(){
  unsigned int * pageTable = (unsigned int *) 0x28000;
  unsigned int * tableDeDirecciones = (unsigned int *) 0x27000;
  *tableDeDirecciones = (unsigned int) pageTable | 3;
  tableDeDirecciones++;
  int i = 1 ;
  for(i = 1; i < 1024; i++){
    *tableDeDirecciones = 0;
    tableDeDirecciones++;
  }
  int j = 0 ;
  for(j = 0; j < 1024; j++){
    *pageTable = j*4096 | 3;
    pageTable++;
  }
}
```

Básicamente lo que se realizó es inicializr una PDE (directorio de páginas) en la dirección 0x27000 y completar su primera entrada con la dirección de una PTE (tabla de páginas) y con los atributos necesarios de supervisor para que sólo pueda ser accedida por el kernel. El resto de las 1024 entradas de la PDE fue completada con ceros ya que no van a ser utilizadas. La PTE fue completada para que mapee con identity mapping páginas de 4 KB, por lo que se completaron cada una de ellas con una dirección base múltiplo de 4096 y nuevamente con los atributos necesarios para que sólo puedan ser accedidas por el Kernel. La función mmu_inicializar_dir_kernel es llamada desde kernel.asm mediante un call.

c) Este otro punto es otro de los más importantes para el funcionamiento del sistema. Desde el kernel lo que se hizo fue inicializar el registro de control CR3. Como no es posible trabajar con el registro CR3 directamente, se copia en eax, el valor 0x27000 que es donde se definió en el punto anterior la PDE. Se copia el valor del registro eax a CR3 y se activa la paginación seteando el bit correspondiente en CR0 con un procedimiento similar al anterior. Para completar el proceso se hace un call a la función mmu_inicializar definida en mmu.c cuyo único propósito es setear una variable global que define la próxima página libre que puede ser utilizada por el sistema.

d) Ejercicio muy similar al de pintar pantalla sólo que además de un color se copian a memoria los valores ASCII del texto correspondiente al nombre de nuestro grupo.

1.4. Ejercicio 4

a) La rutina de inicializacion de mmu simplemente setea un contador llamado 'proxima_pagina_libre' el cual simplemente indica la dirección de memoria en donde comienzan las páginas libres. Luego, cada vez que se necesite una dirección de página libre, se llama a la función mmu_proxima_pagina_fisica_libre la cual se encarga de actualizar la variable antes mencionada y devuelve su valor anterior.

b) En este punto se solicita que se desarrolle una rutina que inicialice el directorio de páginas y la tabla de páginas para una tarea. Cada tarea en este sistema tiene su directorio y tabla de páginas. Se solicita además que la rutina copie el código de la tarea a su área asignada dentro del mapa. Se implementó para cumplir con lo requerido por el enunciado la función mmu_inicializar_dir_tarea la cual recibe como parámetros el CR3, la dirección física de la tarea, y las coordenadas x e y de la posición de la tarea en el mapa. Se valida que la posición sea válida. Luego se obtiene una página nueva llamando a la función mmu_proxima_pagina_fisica_libre para la PDE de la tarea y otra para la PTE de la tarea. La primera entrada de la PDE es la única que se va a usar en la tarea por lo cual se la configura con los atributos correspondientes. El resto de las entradas se completan con cero. Las 1024 entradas de la PTE se completan primero con identity mapping. Luego se llaman a las funciones de mapear pagina tarea y mapear pagina que serán explicadas en el punto que sigue. La función retorna un nuevo CR3 que es simplemente la dirección de la PDE creada anteriormente.

c) Se pide realizar dos rutinas, una que realice el mapeo de páginas y otra el desmapeo de páginas. La función mmu_mapear_pagina implementada recibe como parámetros una dirección virtual, un cr3 y una dirección física. Lo primero que realiza es descomper la virtual para obtener el índice en la PDE y el índice de la PTE. Obtiene la dirección efectiva de la PDE usando el valor base contenido en el cr3 con el offset del indice de la PDE. Si la pagina no está presente, es decir, no hay cargado nada en dicha entrada en la PDE, se crea una nueva página y se setea la misma con permisos de lectura-escritura.

El resto de las entradas se deja en cero en caso de haber sido creada la primera entrada en el PDE. Finalmente se obtiene la PTE usando la dirección base de la entrada de la PDE (sólo utiliza los 20 bits más significativos) con el offset del indice de la PTE. En dicha página se carga la dirección física con los atributos de lectura-escritura y el bit de presente. La función mmu_unmapear_pagina implementada recibe como parámetros una dirección virtual y un cr3. La misma hace un procedimiento similar al mapear pagina: descompone la dirección virtual y obtiene los índices de PDE y PTE. Obtiene la PDE usando la dirección base del registro cr3 y el offset del índice del PDE. Luego con los 20 bits más significativos de la dirección base contenida en la entrada de la PDE y el offset del índice de la PTE se recupera la PTE y en la misma se le configura el bit de presente en cero. Finalmente se llama a la función tblflush() que se encarga de limpiar el caché de traducciones para que el cambio sea actualizado en la tlb.

d) No haremos mucha descripción de este punto ya que como lo indica el enunciado, no debía estar implementado en la solución que se entrega con este informe.

1.5. Ejercicio 5

a) Para este ejericio se agregaron tres entradas en la IDT. Una para la interrupción de reloj, otra para la de teclado y una para la interrupción 0x66, o 102 en decimal (que es con el nombre con el cual quedó configurado). Esto se realizó en los archivos correspondientes a la idt, es decir, idt.h e idt.c.

b) En la rutina de atención de interrupción de reloj se implementó solamente una base que solamente se encargaba de llamar a la función screen_proximo_reloj la cual ya estaba implementada por la cátedra y se encargaba de dibujar en pantalla cómo iba avanzando el reloj.

c) En la rutina de atención de interrupción de teclado se implementó en ASM el detector de teclas presionadas en las cuales imprimiría en pantalla en la esquina superior qué tecla fue y el color representando al jugador que le pertenece esa tecla en los casos de que la misma sea una que deba realizar alguna acción en el juego (tales como w, a, s, d o Lshift). Además, se agregó que el juego no iniciaría hasta que se presione la barra espaciadora, la cual indica el comienzo del juego una vez presionada y en la RAI para esa tecla se llaman a las funciones correspondientes que inicializan el juego.

d) Tal como se pedía en el enunciado, la primera versión de la rutina de atención de la interrupción 0x66 lo único que hacía era escribir el valor 0x42 en el registro eax.

1.6. Ejercicio 6

a) En este ejercicio, se agregaron 47 entradas en la GDT definidas como TSS. La primera para la tarea inicial, la segunda para la tarea Idle y las otras 45 se repartieron 15 por cada jugador, es decir 15 para el jugador A, 15 para el B y 15 para las tareas sanas. Esto se decidió así ya que si bien cada jugador puede lanzar 5 tareas a la vez, en total podría lanzar un máximo de 15, y sería más fácil luego encontrar la próxima entrada libre. Esto sería simplemente tener una variable que se inicialice en 9 (ya que las primeras 4 entradas son nulas, las siguientes 4 entradas son para la segmentación explicada en este mismo informe y las entradas 7 y 8 eran para las tareas mencionadas) y cada vez que se pida la próxima entrada libre, es simplemente dar el valor actual de esa variable y actualizarla sumándole 1.

b) Al inicializar la tss de la tarea idle se agregó la inicialización de la variable que indicará cuál es la próxima entrada libre de la gdt, de la tarea inicial y de una variable que indica cuál es la próxima estructura de tss libre (esto se explica en detalle en el siguiente inciso). En cuanto a la información de la tarea idle, primero se cargó la tss y luego la entrada de la gdt de su respectivo descriptor. En la TSS, se cargaron los segmentos (excepto el de código) con el selector de segmento de gdt 0x30, el cual indica el segmento de datos de nivel 0, mientras que el segmento de código se cargó con 0x20, o sea el segmento de código de nivel 0. En cuanto a la pila, se indicó la dirección que se solicitó en el enunciado (0x27000). Para el cr3, nuevamente como indicaba el enunciado, se cargó el cr3 del kernel. Por último, para el eip, se colocó también la dirección informada en el enunciado (0x10000). En la GDT se configuró el selector de TSS con la dirección y tamaño de la tss correspondiente, el dpl se colocó en 0 (el rpl en la tss se cargó en 0) y al ocupar un tamaño menor a 1mb (la tss), se colocó el bit de G en 0.

c) Para la función solicitada en este ejercicio, se reciben por parametros la dirección física original de la tarea, un *x* y un *y*, que son las posiciones que ocuparán en el 'mapa', y devuelve el número de entrada de la gdt que ocupa el descriptor de tss de la nueva tarea. Para poder tener una tss diferente en cada tarea, se crearon 45 tss distintas al principio del archivo tss.c y luego se colocaron en un Array de 45 posiciones, las cuales cada una tiene la dirección de memoria donde está cada tss. De esta forma, se puede obtener una nueva tss cada vez que se la solicite, ya que es cosa de ver la siguiente posicion en el Array y así obtener la dirección de memoria de la nueva tss. De la misma forma que para la tarea

Idle, se carga la gdt con el descriptor de tss correspondiente a una tarea de nivel 3, y se inicializa su tss. La TSS queda cargada de la siguiente forma:

```
nueva_tss->cs = GDT_OFF_IDX_DESC_CODE3 + 3;
nueva_tss->es = GDT_OFF_IDX_DESC_DATA3 + 3;
nueva_tss->ss = GDT_OFF_IDX_DESC_DATA3 + 3;
nueva_tss->ds = GDT_OFF_IDX_DESC_DATA3 + 3;
nueva_tss->fs = GDT_OFF_IDX_DESC_DATA3 + 3;
nueva_tss->gs = GDT_OFF_IDX_DESC_DATA3 + 3;
nueva_tss->esp = 0x08000000 + 0xfff;
nueva_tss->ebp = 0x08000000 + 0xfff;
nueva_tss->eflags = 0x202;
nueva_tss->eip = 0x08000000;
unsigned int nuevaCR3 = mmu_inicializar_dir_tarea(cr3, dirFisicaTareaOriginal, x, y);
nueva_tss->cr3 = nuevaCR3;
nueva_tss->esp0 = mmu_proxima_pagina_fisica_libre();
nueva_tss->ss0 = GDT_OFF_IDX_DESC_DATA0;
nueva_tss->iomap = 0xFFFF;
```

Donde nueva_tss es el puntero a la dirección de memoria de la nueva tss. GDT_OFF_IDX_DESC_CODE3 es el selector de segmento de datos de nivel 3, y se le suma 3 cuando corresponde para setear el RPL en 3. Esp y ebp se cargan con esa dirección ya que es donde debe estar mapeada la tarea y se le suma 0xfff por ser la pila de la misma. Por la misma razón que esp y ebp, eip se coloca 0x08000000 pero no se le suma 0xfff por no ser la pila sino el instruction pointer. Para el cr3 se llama a la función mmu_inicializar_dir_tarea explicada con anterioridad en este informe. mmu_proxima_pagina_fisica_libre() se utiliza para obtener una nueva pila y GDT_OFF_IDX_DESC_DATA0 es el selector de segmento de datos de nivel 0. Además, al llamar a esta función, se agregó que se llame a la función cargarTareaEnCola la cual se encarga de cargarla en el scheduler (más información de esta tarea en el ejercicio 7).

d) Este punto se realizó en la función que inicializa la tss de idle. La misma fue cargada con la misma información que para la tss de idle pero con la tss de la tarea inicial.

e) Descripto en el inciso (b)

f) Al inicializar las tss y la gdt para las tareas idle e inicial, somo se mencionó en el inciso (b) se obtiene el selector de segmento de la gdt que, en este caso, es el de la TSS de la tarea inicial. Por lo tanto, luego de llamar a la funcion que se encarga de inicializar a esas dos tareas, tenemos en eax el selector mencionado y se carga en el Task Register mediante la instrucción ltr. Para saltar a la tarea Idle, se le suma 8 a eax (para obtener el siguiente selector) y se pushea el registro para usarlo luego de inicializar las otras estructuras necesarias. Para hacer el salto, dejamos en eax el selector de segmento de la TSS de Idle, y se realizan las siguientes instrucciones:

```
mov [selector], ax
jmp far [offset]
```

siendo [selector] y [offset] espacios definidos en memoria mediante el uso de dw y dd respectivamente.

1.7. Ejercicio 7

a) La inicialización del scheduler se encarga de setear las variable correspondientes que serán funcionales al juego. Estas son 3 int (proximoColaA, proximoColaB y proximoColaNadie) que indicarán la próxima posicion del arreglo de tareas de cada jugador en la cual se almacena mediante el struct Tarea información sobre la misma. Este arreglo es en realidad una matriz de 3 x 15, en la cual en cada fila está el arreglo de 15 posiciones de Tareas. Cada una de ellas representa el arreglo de 15 posiciones de tarea de cada jugador. Para el caso del jugador A y B, nos ocupamos de que no se escriba ninguna tarea más allá de la posicion 5 ya que cada jugador solo puede tener hasta 5 tareas al mismo tiempo. Se setea también la variable colaActual, que indica cual de los 3 arreglos de la matriz es al que se deberá acceder para buscar la siguiente tarea. El puntero a Tarea, tareaActual, apunta a la estructura Tarea en la matriz jugadores que contiene la información de la tarea que se está ejecutando en ese momento. Por último, siguienteIndiceDeTareaEnCola contiene el próximo índice a seleccionar de cada arreglo de la matriz jugadores, teniendo en comun que cada número de posicion del arreglo siguienteIndiceDeTareaEnCola equivale al número de posicion del arreglo en la matriz. El struct Tarea está compuesto de la siguiente forma:

```
typedef struct str_tarea{
  tupla posicion;
  tupla posicionOriginal;
  unsigned short indiceGdt;
  char presente;
  unsigned int cr3Actual;
  int dueno;
  int duenoOriginal;
  int relojPropioX;
  short posReloj;
} tarea;
```

donde el struct tupla está conformado así:

```
typedef struct str_tupla{
  unsigned short x;
  unsigned short y;
} tupla;
```

En el struct tarea, posicion representa la posicion actual de la tarea. posicionOriginal la posición donde se lanzó originalmente (para el caso de las tareas sanas, el lugar donde se inicializó). indiceGdt es lo que el nombre indica. el char presente es para indicar si está presente en el scheduler o no (se setea en 0 cuando se desaloja la tarea). cr3Actual es el cr3 que utiliza la tarea. dueno es el número que representa al jugador que le pertenece la tarea, que tanto para este como para duenoOriginal es 0 si es el jugador A, 1 jugador B y 2 tarea sana. duenoOriginal es el jugador que lanzó originalmente la tarea (este valor no cambia una vez que se setea para cada tarea). relojPropioX es la posición en X en la que se va a dibujar el reloj de esa tarea. posReloj es el número que indica cual es el siguiente dibujo que representa al reloj a imprimir en pantalla.

b)

f) En este punto se solicita implementar la funcionalidad de debugging. Lo que se realizó fue modificar el código de la interrupción del reloj para que se fije si está habilitado el debug y está pintada la pantalla no se salte entre tareas. Se agregaron dos variables globales, una que indica si el debug está habilitado o no y otra que indica si la pantala de debug está pintada o no. En todas las rutinas de atención de interrupciones lo que se hizo es hacer un call a una función que devuelve cero si el debug no está activado, en cuyo caso continua con la ejecución de la atención de la rutina, y si devuelve 1 (significa que el debug está activado) lo que hace es pushear todos los registros que se deben mostrar en pantalla cuando ocurre una excepción, algunos de los cuales son extraídos de la pila del handler de exepción y llama a una función atenderDebug cuyo propósito es mostrar por pantalla la información del debug.

1.8. El retorno de la GDT y la aparicion de la TSS

1.9. El Scheduler

Para realizar el scheduler, decidimos crear un struct tarea que contiene la posicion en el mapa,su indice en la GDT, su cr3, si tiene prendido el bit de presente en la GDT

2. Conclusiones y trabajo futuro