DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico III

## Asimilación Cronenberg

Organizacion del Computador II

Primer Cuatrimestre del 2020


| Integrante | LU | Correo electrónico |
|---|---|---|
| Morán, Martín | 650/17 | martinmoran1994@gmail.com |
| Sosa, Patricio | 218/16 | patriciososa91@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

# 1. Informe

En este informe trataremos los pasos que seguimos para lograr el correcto funcionamiento del sistema que ejecuta el juego Asimilación Cronenberg. Para conseguir una lectura organizada y fácil de seguir dividiremos el informe en secciones dedicadas a cada uno de los ejercicios del trabajo práctico.

## 1.1. Ejercicio 1

En este ejercicio inicializamos la Global Descriptor Table con 4 segmentos: 2 para código y datos de nivel kernel (DPL 0) y 2 para lo código y datos de nivel usuario (DPL 3). Sus posiciones en la GDT serán 9 y 11 para segmento de código para kernel y usuario respectivamente,además las posiciones 8 y 10 son del segmento de datos para los mismos. Como estamos trabajando con el modelo de segmentación flat los 4 segmentos abarcan desde la base en 0x0 hasta el límite de 137 MB. En cuanto a los atributos damos las siguientes descripciones:

- P $= 1$: Damos a entender que el segmento se encuentra presente en RAM.
- S $= 1$: Estos 4 segmentos van a ser o de datos o de código, no de sistema.
- Type:
  - 0xa ($= 1010$) para segmentos de código. Habilitamos el bit R para la posterior lectura del código. Bit C en 0 porque queremos que su privilegio sea estricto y no ajustable.
  - 0x2 ($= 0010$) para segmentos de datos. Bit ED en 0 porque no lo vamos a utilizar como pila y bit W en 1 porque nos interesa escribir información en el más adelante.
- L $= 0$: Vamos a trabajar en modo de 32 bits.
- G $= 1$: Como nos piden MBytes necesitamos este bit en 1.
- DB $= 1$: Ídem bit L.
- DPL:
  - 0 para los segmentos del kernel.
  - 3 para los segmentos de usuario.

Además de estos 4 segmentos, inicializamos uno más para el area de la pantalla en memoria en la posicion 12. El mismo empieza en la dirección 0xB8000 y tiene un tamaño de 4 KBytes, por lo que el bit G va estar en 0. Los demás bits quedan como el segmento de datos ya descripto del kernel.

Luego de definir estas estructuras, en el código del kernel habilitamos el address 20 (A20), cargamos la GDT usando su descriptor mediante la instrucción lgdt. Con esto hecho, habilitamos el PE del registro CR0 para habilitar el modo protegido y hacemos un JMP far para pasar a trabajar en este modo. Aquí, cargaremos los selectores de segmentos a estos y la base de la pila a los registros del stack esp y ebp. Luego, imprimiremos el mensaje de bienvenida en la mitad de la pantalla y pintaremos el sector del tablero del juego (que ocupa 80x40 bytes) usando el último segmento definido.

## 1.2. Ejercicio 2

En este ejercicio inicializaremos la Interrupt Descriptor Table. Para este trabajo solo utilizaremos descriptores de tipo Interrupt Gate. Definimos las interrupciones número 0 a 19. Para cada uno definimos el selector de segmento en 0x48 para que puedan acceder a al descriptor de las rutinas en la GDT localizada en el segmento de código del kernel (podría ser también la LDT pero no la usamos en este trabajo) y obtener la dirección base de la misma. Sus offsets combinadas con la dirección base se corresponden con su posición en el sector de código de las rutinas de interrupciones. Los atributos los setearemos en 0x8E00 para que se correspondan con los siguientes bits:

- P = 1: Como siempre, queremos que esté presente.
- S = 0: Ya que estos descriptores serán usados por el sistema.
- DPL = 0: Estas interrupciones solo serán producidas por el kernel.

Los siguientes 4 bits se corresponden con el tipo de una Interrupt Gate de 32 bits. El ultimo byte lo dejamos en 0 ya que no le damos importancia.

En la sección de atención de rutinas de interrupción definimos el código necesario para que imprima el número de interrupción por pantalla y se interrumpa la ejecución. Cada vez que se produce una interrupción preservamos siempre el valor de los registros con la instrucción pushad y los restauramos al final con popad

En el código del kernel, llamaremos a la función idt_init() que inicializa los descriptores mencionados anteriormente con sus respectivos valores. Luego, utilizaremos la instrucción lidt para cargar la IDT y que el procesador pueda empezar a usarla.

## 1.3. Ejercicio 3

Definimos en este ejercicio las entradas de la IDT correspondientes con clock y el teclado. Las mismas se corresponderán con la interrupción 32 y 33 respectivamente. Además de estas interrupciones, definimos las interrupciones 137, 138 y 139. El descriptor de estas 3 ultimas tendrán los mismos atributos, salvo por la única diferencia DPL = 3 ya que podrá ser llamada por el usuario.

En la rutina de interrupción del clock simplemente llamamos a la función nextClock la cual se encarga de mover la animación en la esquina inferior derecha de la pantalla.

Para la rutina de interrupción del teclado, obtendremos el número de la tecla presionada. Si la misma era menor a la correspondiente con el 0 salimos de la interrupción. Lo mismo con la correspondiente a la 9. Si se encuentra en el rango, ejecutamos el código necesario para imprimir la tecla en la parte superior derecha de la pantalla.

Para las interrupciones 137, 138 y 139 se modifico el registro eax como se pide en el enunciado. Quedaron definidas para su posterior uso.

## 1.4. Ejercicio 4

En el presente ejercicio inicializamos el directorio de páginas y sus páginas para que el kernel lo use. El mismo estará localizado en la dirección 0x27000. Uno podría pensar que como tiene la misma dirección que la pila del kernel podría haber problemas de sobre escritura pero la pila crece en dirección contraria al directorio por lo que no hay problema. Como todo directorio, tendrá 1024 posiciones de 32 bits correspondientes con el tamaño de una página (1024*32 = 4096 = tamaño en bytes de una página). Lo mismo con las tablas. En primera instancia, al inicializarlo, dejaremos todas las posiciones del directorio en 0. Luego, procedemos con inicializar el primer descriptor del directorio para mapear las direcciones de memoria de 0x00000000 a 0x003FFFFF. A este descriptor de tabla le pondremos:

- Su dirección base en 0x28000 (dirección donde empiezan las tablas) shifteado 12 veces a derecha por diseño de la estructura.
- Los bits disponibles y G los dejamos en 0 ya que no nos interesan.
- El bit PS en 0 porque el tamaño de nuestras páginas es de 4KB.
- Los bit A, PCD y PWT en 0 porque tampoco nos interesan.
- Bit U/S en 0 correspondiente con el kernel.
- R/W en 1 porque vamos a querer tanto leer como escribir en la página.

Para la tabla, como la posición base de la sección de tablas de páginas se corresponde con la con la tabla inicial no es necesario sumarle algún offset. Ahora inicializamos todas los descriptores de la tabla con los siguientes valores de bits:

- disponibles, G, A, PCD, PWT, R/W, U/S y P iguales al del directorio.
- PAT $= 0$: No nos interesa.
- Dirty $= 0$: Tampoco nos interesa.

A cada descriptor de página le asignamos su dirección base que corresponda con su número de página. De esta manera, quedan mapeadas con Identity Mapping las direcciones de 0x00000000 a 0x003FFFFF.

Para habilitar paginación, en el código del kernel cargamos la dirección del directorio de tablas en el Registro de Control 3 (CR3). Luego, seteamos el bit de paginación del Registro de Control 0 en 1.

## 1.5. Ejercicio 5

Para administrar la memoria del área libre del kernel, inicializaremos una variable global próxima_pagina_libre_kernel. La misma tomará el valor inicial del área libre 0x100000. Cada vez que se requiera una nueva página llamaremos a la función mmu_nextFreeKernelPage() y se le sumará a nuestra variable el tamaño de una página para poder usar la próxima página libre.

Para inicializar las estructuras necesarias para la memoria de una tarea, primero reservamos una página para el directorio de tablas para las tareas. Luego pedimos dos páginas libres del área de tareas, calculadas utilizando los parámetros 'X' e 'Y', de modo que coincidan la posición en el mundo de la tarea con el lugar que ocupa en memoria. Para ello utilizaremos la fórmula $INICIO\_DEL\_MUNDO + Y * TAMAÑO\_FILA * 2 * PAGE\_SIZE + X * 2 * PAGE\_SIZE$ con INICIO_DEL_MUNDO = 0x400000, TAMAÑO_FILA = 80 y PAGE_SIZE = 4KB. La idea detrás de esta fórmula es tratar al mapa de memoria del mundo y de la pantalla como una matriz, siendo el parámetro 'Y' la fila y 'X' la columna, mientras que el tamaño de cada posición está dado por 8KB (1 página de código + 1 página de pila). Al directorio de tablas lo inicializamos con todos sus descriptores en 0. Primero mapeamos el kernel (4MB) con Identity Mapping en el nuevo directorio. Después, mapeamos la dirección virtual del código y la pila de la tarea (0x8000000) con la dirección física de las nuevas páginas en el directorio nuevo. Como conocemos la dirección física podemos copiar el código de la tarea a esa dirección. Pero para esto necesitamos mapear las paginas nuevas en el directorio actual, usamos Identity Mapping para asegurarnos de no pisar otro mapeo de ese directorio. Copiamos tanto el código como la pila. Debemos desmapear las dos páginas en el directorio actual, para que solo la tarea pueda acceder a sus páginas. Finalmente, devolvemos el nuevo directorio con el código ya copiado.

Podemos observar un esquema del proceso del copiado de código en la figura 1 y otro esquema que refleja el estado, tanto de la memoria como del nuevo directorio ya mapeado, en la figura 2 .

Para realizar los mapeos procedemos con el método establecido: Le pasamos a la función de mapeo la dirección física y virtual y el CR3 del directorio de páginas a mapear. En este caso, también le pasamos los bit U/S y R/W mediante el parámetro attrs para setearlos (los descriptores de tablas siempre los setean en 1, así cada pagina se maneja con sus atributos). Con los 10 bits más significativos de la dirección virtual, obtenemos el índice del descriptor de la tabla en el directorio. El descriptor nos indica la base de la tabla con los descriptores de páginas. Si está en 0 el bit P, pedimos una nueva página para crear una nueva tabla e inicializamos sus valores en 0 y cargamos su descriptor en el directorio con la base y atributos correspondientes. Con los siguientes 10 bits de la dirección virtual, obtenemos el índice de la entrada de la tabla que buscamos. Usamos la dirección base de la tabla y, en conjunto con el índice, accedemos al descriptor de la página que buscábamos. Modificamos este descriptor con los bits U/S y R/W pasados por parámetro. Seteamos el bit P en 1 y le asignamos a la base de la página la dirección física pasada por parámetro shifteado 12 a derecha.

Para el desmapeo, seguimos un proceso similar, pero esta vez solo seteamos el bit P en 0, con lo que ya no estará presente la pagina (decidimos dejar las tablas presentes en el directorio aunque estas ya no contengan ninguna página presente).

## 1.6. Ejercicio 6

Para la tarea inicial y la tarea idle tenemos la función gdt_tss_init que inicializa los índices 13 (Tarea Inicial) y 14 (Tarea Idle) en la gdt en el archivo gdt.c, a estos descriptores le asignamos como base el puntero a la tss idle y la tss inicial respectivamente que están definidos en tss.h, y los siguientes atributos:

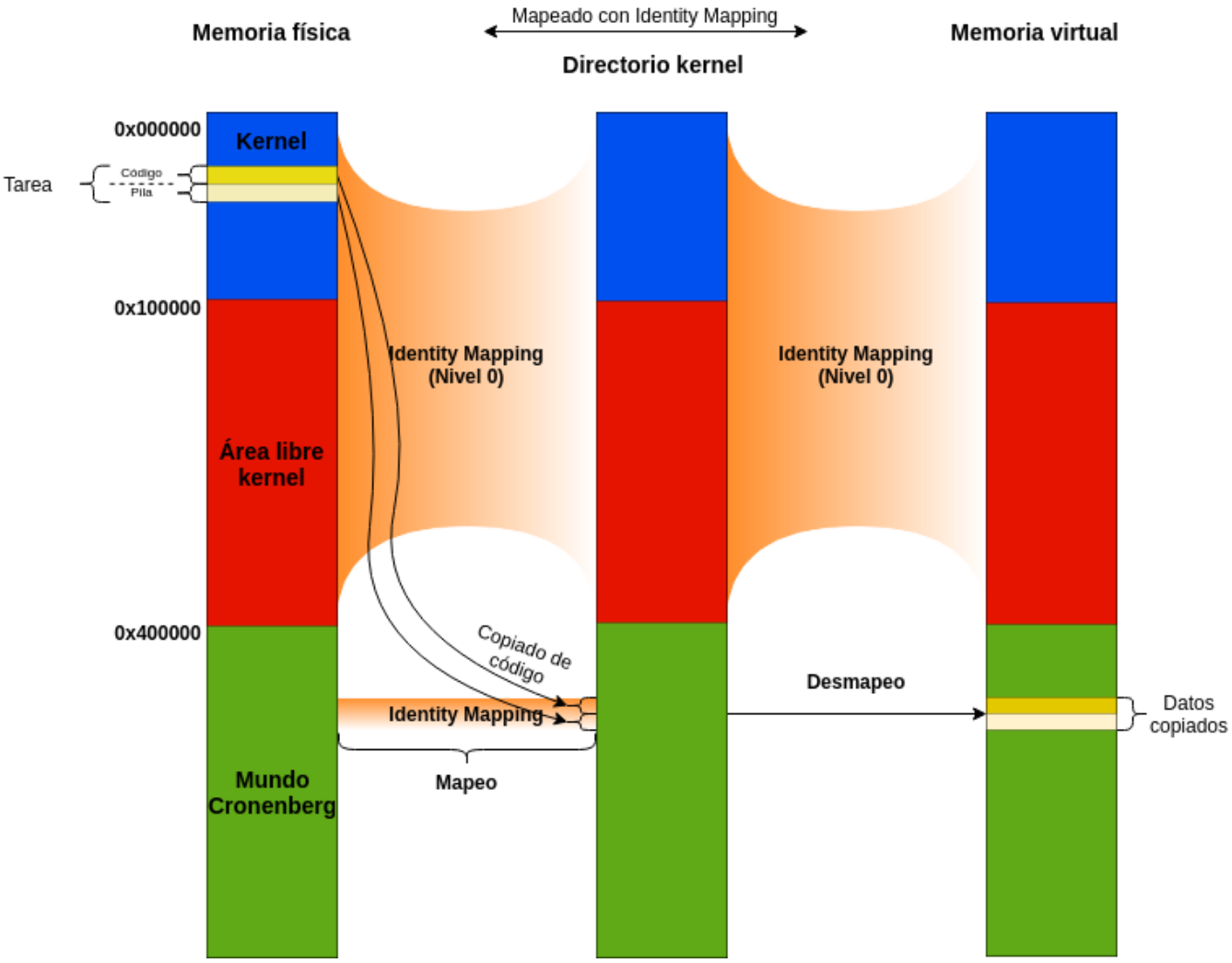


Figura 1: Mapeo, copia y desmapeo del código de una tarea

- P = 1: el descriptor se encuentra presente .
- S = 0: Los descriptores de TSS los maneja el sistema.
- Type: 0x9 (= 1001). Con el bit S en 0 esto define un descriptor de TSS para 32 bits.
- L = 0: Vamos a trabajar en modo de 32 bits.
- G = 0: El límite de las TSS es el tamaño de su estructura.
- DB = 1: Ídem bit L.
- DPL = 0: Los descriptores de TSS tienen que tener nivel 0.

Fue necesario escribir esta función ya que los valores que toman las bases de estos descriptores es necesario obtenerlos en tiempo de ejecución. Además, creamos la función gdt_tss() para definir los descriptores de las tareas restantes. Para todos estos descriptores fijamos el límite en 0x67 (tamaño de una TSS). Todos con privilegio en 0 (DPL=0), bit de sistema en 0 y tipo 9. Definimos sus direcciones base referenciando sus definiciones en el archivo TSS.

A la función encargada de inicializar las futuras TSS de las tareas la llamamos iniciar_tss. La misma, tomará como parámetros la dirección del código fuente de la tarea, el puntero a un struct TSS a ser completado y las coordenadas X e Y en donde ubicar a la tarea como mencionamos en el ejercicio 5. A los diferentes campos de la TSS los llenamos con ayuda de los siguientes defines:

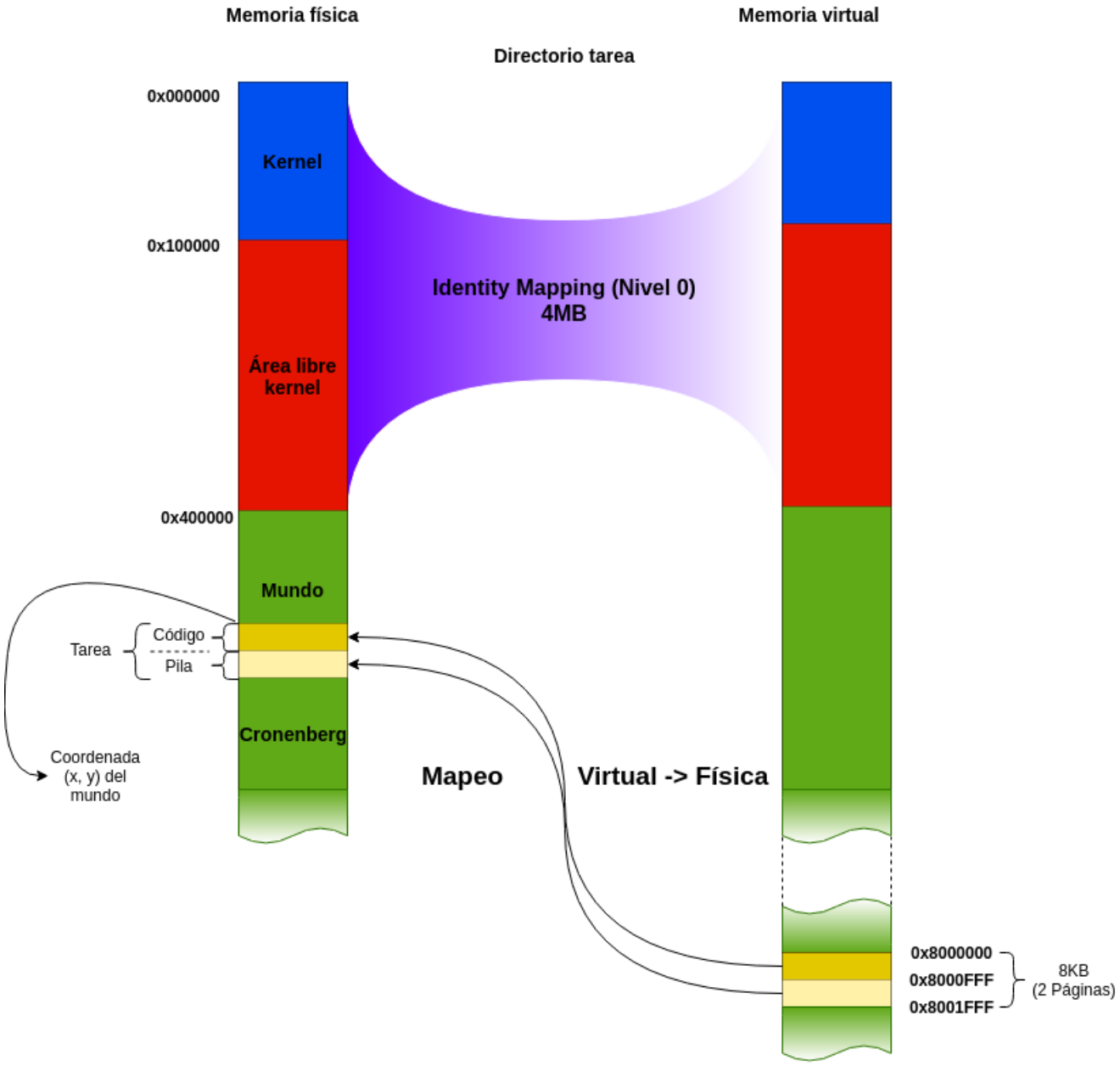


Figura 2: Esquema post mapeos y copiado de una tarea

- TASK_CODE = 0x8000000
- PAGE_SIZE = 0x1000
- CS_CODIGO_USUARIO = 0x005B
- DS_DATA_KERNEL = 0x0040
- DS_DATA_USUARIO = 0x0053

Completamos los campos para cada una de las tareas de la siguiente manera:

- EIP = TASK_CODE: dirección virtual a ejecutar la tarea.
- CR3: Creamos un nuevo directorio como describimos en el ejercicio 5.
- ESP = TASK_CODE + 2*PAGE_SIZE: Dirección virtual de la pila. Como la pila crece hacia direcciones menores, colocamos el puntero al final de la página de la pila.

- ESP0 = mmu_nextFreeKernelPage() + PAGE_SIZE: Pedimos una página del área libre del kernel para colocar la pila de nivel 0. Al igual que el ESP, colocamos el puntero al final de la página.
- SS0 = DS_DATA_KERNEL: Utilizamos el segmento de datos del kernel para poder utilizar la pila de nivel 0 luego del cambio de nivel de privilegio durante una interrupción.
- CS = CS_CODIGO_USUARIO: Selector de segmento de código de nivel 3.
- DS, SS, ES, DS, FS, GS = DS_DATA_USUARIO: Las tareas corren a nivel 3 por lo que utilizan el segmento del mismo nivel.
- EBP: Ídem ESP.

## 1.7. Ejercicio 7

### 1.7.1. Scheduler

Para iniciar nuestra estructura de Scheduler tenemos la función sched_init en sched.c. Para controlar el schedule propiamente dicho, nos valdremos de un arreglo de 8 posiciones, correspondientes con la cantidad de tareas del sistema. En cada posición del mismo tendremos el siguiente struct:

```
typedef struct str_sched
{
uint32_t pos_x;
uint32_t pos_y;
uint8_t is_alive;
uint16_t tss_selector;
}__attribute__((__packed__, aligned (8))) sched;
```

Donde:

- pos_x: Indica la posición horizontal de la tarea. Puede tomar valores entre 0 y 79.
- pos_y: Indica la posición vertical de la tarea. Puede tomar valores entre 0 y 39.
- is_alive: Booleano que indica si una tarea sigue viva. Para todas las tareas se inicializa en TRUE.
- tss_selector: selector de TSS de la tarea.

La función sched_init será la encargará principiar el arreglo de scheduling. Para cada una de las tareas asociadas a los structs en el arreglo, asignaremos un par de coordenadas (x, y) a modo de posición inicial. Cada tarea deberá tener una posición inicial diferente. Valiéndonos de estas coordenadas, la dirección del código fuente de una tarea y el puntero a su respectivo struct TSS asignado para inicializar, llamamos a la función iniciar_tss pasándole los parámetros mencionados. Esto creará la TSS de la tarea, su directorio de tablas con los mapeos necesarios y posicionará a la tarea en el lugar del mundo Cronenberg dado por el par de coordenadas.

Además, durante la creación del Scheduler, inicializaremos las siguientes variables, cuyos propósitos a grandes rangos es asegurar el correcto comportamiento del juego/sistema:

- current_task : Indica cual es la tarea actual que esta corriendo. Inicia con el valor 0, correspondiente a la tarea Idle.
- modo_debug: Booleano que activa o desactiva el modo debug. Se inicializa en FALSE.
- minds_conquered_rickC137: Contador que indica la cantidad de mentes Cronenberg conquistadas por el universo C137. Se inicializa en cero.

- minds_conquered_rickD248: Contador que indica la cantidad de mentes Cronenberg conquistadas por el universo D248. Se inicializa en cero.

- canbeuse_byMortyD248: Booleano que indica si la Syscall usePortalGun (int 137) puede ser utilizada por la tarea MortyD248. Se setea inicialmente en FALSE.

- canbeuse_byMortyC137: Booleano que indica si la Syscall usePortalGun (int 137) puede ser utilizada por la tarea MortyC137. Se setea inicialmente en FALSE.

- portalNotYetCreated_byC137: Booleano que indica si la Syscall usePortalGun (int 137) puede ser utilizada por la tarea rickC137. Se utiliza para verificar que solo sea utilizado una vez por cada tick de reloj. Se inicializa en TRUE.

- portalNotYetCreated_byD248: Booleano que indica si la Syscall usePortalGun (int 137) puede ser utilizada por la tarea rickC137. Se utiliza para verificar que solo sea utilizado una vez por cada tick de reloj, se inicializa en TRUE.

- times_GunUsed_rickC137: Contador que indica cuantas veces fue utilizada la Syscall usePortalGun (int 137) por rickC137.

- times_GunUsed_rickD248: Contador que indica cuantas veces fue utilizada la Syscall usePortalGun (int 137) por rickD248.

- cronenbergs_dead: Contador de la cantidad de tareas Cronenbergs muertas.
- termino_juego: Booleano que se seteara en TRUE cuando termine el juego.
- _portal:

```
typedef struct str_portal
{
int pos_x;
int pos_y;
}__attribute__((__packed__, aligned (8))) _portal;
```

  Struct donde se guarda la posición de cada portal creado por cada universo, se crea 1 variable por cada universo las cuales son:

  - portals_Created_rickC137: Guarda la posición del portal creado por el Rick del universo C137. Se inicializa en con el valor (-1,-1), indican que no sea creo ningún portal todavía.

  - portals_Created_rickD248: Guarda la posición del portal creado por el Rick del universo D248. Se inicializa en con el valor (-1,-1), indican que no sea creo ningún portal todavía.

  - portals_Created_mortyC137: Guarda la posición del portal creado por el Morty del universo C137. Se inicializa en con el valor (-1,-1), indican que no sea creo ningún portal todavía.

  - portals_Created_mortyD248: Guarda la posición del portal creado por el Morty del universo D248. Se inicializa en con el valor (-1,-1), indican que no sea creo ningún portal todavía.

- excepcion: Variable booleana que se seteará en TRUE cuando se produzca una excepción. Se inicializa en FALSE.

La función sched_nextTask se encargará de devolver el selector de TSS de la próxima tarea a ejecutar. Dentro de la función, utilizaremos una variable global llamada current_task, la cual indica el índice del arreglo correspondiente a la tarea actual. En la función, utilizaremos esta variable para buscar la siguiente tarea viva en nuestro arreglo, actualizándola para que indique la nueva tarea en ejecución. Durante la interrupción de clock, llamaremos a esta función para realizar el cambio de contexto (si no es la misma tarea que está corriendo).

### 1.7.2. Excepciones 0-19

Para cada una de las excepciones, pasaremos por pila los parámetros a ser imprimidos en pantalla si el modo debug está activo. También seteamos en TRUE el booleano .$^{ex}$cepcion"que indicando que se produjo una. Luego de imprimir los registros si el modo debug está activo, llamamos a la función killcurrent_task para matar a la tarea que produjo la excepción. Como la tarea no volverá a ejecutarse, no nos preocupamos del estado de su pila de nivel 0.

### 1.7.3. Interrupción del clock (int 32)

Llamamos a la función pic_finish1 para indicar que la interrupción fue atendida. Verificaremos si ya terminó el juego. De ser así, realizaremos directamente iret para volver en todos los clocks a la tarea Idle. Llamamos a la función portalCreatReset para setear en TRUE la variables globales portalNotYetCreated_by que indican si las tareas Rick pueden crear portales. Los llamados a las funciones verificar_excepcion, verificar_modo_debug y activar_desactivar_excepcion tienen que ver con el funcionamiento del modo debug y se explicarán con más detalle en esa sección. Normalmente, llamamos a la función sched_nextTask para obtener el selector de TSS de la próxima tarea a ejecutar, chequear que no sea la misma que la actual y ejecutar el cambio de contexto.

### 1.7.4. Interrupción del teclado (int 33)

Obtenemos la tecla que fue presionada (o liberada). Si fue presionada la "y", cambiaremos el valor del booleano modo_debug para indicar que fue activado/desactivado. Si la tecla presionada fue alguna en el rango 0-9, se printeara su número en la esquina superior derecha de la pantalla. Finalmente, indicamos que fue interrumpida la interrupción y hacemos iret.

### 1.7.5. Servicios de Sistema

Modificamos la rutina de excepciones en isr.asm desde allí llamamos a los servicios que fueron implementados en C en el archivo sched.c, a continuación damos un detalle de las implementaciones de los servicios:

**usePortalGun(Int 137)**

Se trata del servicio utilizado tanto por Rick o Morty para crear portales. Se consideran 3 casos:

1. La current_task es alguna tarea Cronenberg: En este caso se procede a marcarla como muerta en el tablero y luego a matar dicha tarea utilizando la función killcurrent_task().

2. La current_task es una tarea Rick de alguno de los universos: Además se verifica que la tarea Rick pueda utilizar el servicio, chequeando el valor de la variable global portalNotYetCreated_byC137, a continuación se detallan los siguientes subcasos de este branch:

    - cross == 1 : El cual a su vez de subdivide en 2 posibilidades:

- withMorty == 1 : En primera instancia se procede a verificar el Morty asociado al Rick llamante para obtener la dirección del código fuente de dichas tareas. Luego se calcula la posición de destino de cada tarea utilizando los desplazamientos x e y recibidos como parámetros. Luego, se obtiene el cr3 de ambas tareas. Para obtener el de la tarea Rick, como sabemos que es la actual, simplemente utilizamos la función rcr3 que devuelve el cr3 de la tarea actual. Para el de Morty hacemos lo siguiente: tomamos su selector de TSS del arreglo de Scheduling. Con el mismo, obtenemos el índice en la GDT del descriptor, extraemos la dirección de su TSS y finalmente de la TSS su cr3. Con el cr3 actual, utilizamos Identity Mapping con las nuevas direcciones físicas para copiar el código fuente de ambas tareas a sus nuevas posiciones. Una vez copiadas las tareas se procede a deshacer los mapeos recién (su única función era poder copiar el código). Una vez hecho esto, el código esta copiado en la nueva posición de la memoria física pero el mapeo virtual sigue siendo el viejo. Para cada respectivo cr3, tomamos las direcciones virtuales del código y la pila y las mapeamos a las nuevas direcciones de la memoria física. Con esto queda realizado el traslado de código y remapeo a memoria física. Por ultimo, recalculamos las nuevas posiciones de las tareas y las actualizamos en la estructura del Scheduler y llamamos a la función reset_screen(), con lo cual actualizamos la pantalla con las nuevas posiciones. Luego, actualizamos los variables globales correspondientes a portalNotYetCreated_by (la seteamos en FALSE) asociada a la tarea Rick que utilizo el servicio, indicando que se creo un portal y evitara que el mismo Rick cree otro portal hasta el siguiente tick de reloj (durante en el cual se seteara en TRUE), el contador times_GunUsed_ además se verifica si dicho contador llego a 10 para habilitar al Morty correspondiente a usar la syscall mediante la variable canbeuse_byMorty.

- withMorty == 0: En este caso solo cruza Rick, se siguen casi los mismos pasos que el caso anterior solo que únicamente utilizamos los datos asociados a Rick.

- cross == 0: Solo se procede a mapear el área del portal de la tarea. También se subdivide en 2 casos:

  - withMorty == 1: Mapeamos el portal de la tarea Rick utilizando el desplazamiento desde la posición de la tarea Morty asociada. En primera instancia se procede a verificar de que universo proviene la tarea Morty para obtener las coordenadas de dicha tarea. Con ellas, se calcula la dirección de destino utilizando el desplazamiento recibido como parámetro, se obtiene el cr3 dela tarea Rick usando rcr3 y mapeamos las paginas del portal con las de destino en la memoria física. Por ultimo, actualizamos las posiciones del portal en las variables globales para tal fin y luego actualizamos los variables globales correspondientes a portalNotYetCreated_by, indicando que se creo un portal, que luego se actualizara una vez que se ingrese al siguiente tick de reloj, el contador times_GunUsed_ además se verifica si dicho contador llego a 10 para habilitar al Morty correspondiente a usar la syscall. En ultima instancia, se resetea la pantalla llamando a la función reset_screen().

  - withMorty == 0: Mismo procedimiento que el caso anterior, solo que esta vez solo se mapea el portal de la tarea Rick usando su posición actual para calcular el desplazamiento.

3. La current_task es una tarea Morty de alguno de los universos: Para este caso, ignoramos el parametro withMorty. Además, se verifica que la tarea Morty pueda utilizar el servicio, chequeando el valor de la variable global canbeuse_byMorty. A continuación, se detallan los siguientes subcasos de este branch:

   - cross == 1: Solo se remapea la tarea Morty. Se procede de la misma manera que el caso cross==1 de la tarea Rick detallada mas arriba pero solo para una tarea individual, en este

caso la Morty que llamo a la syscall.

- cross == 0: Solo se mapea el área del portal de la tarea Morty. Es en esencia el mismo procedimiento que la cross==0 de la tarea Rick detalladas mas arriba pero en este caso la Morty que llamo a la syscall

Para cada uno de los casos descriptos el valor de retorno será 1, indicando que fue creado el portal. Si la tarea es una Rick que ya creo un portal en el clock actual o una Morty que todavía no puede utilizar el suyo, devolveremos 0 indicando que no se creó el portal.

**IamRick(int 138)**

Se trata del servicio que indica de que universo proviene una tarea indicando el código del universo a través del parámetro que recibe la syscall y así poder agregarla a la lista de conquistados de cada universo. Una vez verificado que la tarea que la llamo es una Cronenberg, se divide en 2 casos:

- code == 0xC137: Se aumenta la variable global minds_conquered_rickC137, se printea en pantalla el nuevo valor y por ultimo se marca en el tablero la nueva conquista y se procede a matar a la tarea utilizando la función killcurrent_task().
- code == 0xD248: Se aumenta la variable global minds_conquered_rickD249, se printea en pantalla el nuevo valor y por ultimo se marca en el tablero la nueva y se procede a matar a la tarea utilizando la función killcurrent_task().

**whereIsMorty(int 139)**

Se trata del servicio que indica el desplazamiento a realizar para que la tarea Rick que utilizó el servicio llegue a la tarea Morty correspondiente. Para este servicio se cuenta con 2 funciones que realizan el mismo procedimiento para cada coordenada en particular. Se procede a llamarlas desde asm y se guardan ambos valores en la pila para finalmente devolverlos en los registros correspondientes. A continuación, se detalla el funcionamiento para ambas:
Primero se inicializa una variable que representa el desplazamiento en al coordenada correspondiente, luego se verifica que tarea hizo la llamada a la syscall. Para el caso de que sea una tarea Rick, se verifican las posiciones correspondiente a su Morty, se calcula el desplazamiento y se devuelve dicho valor. En el caso de ser una tarea Cronenberg, se marca en el tablero como muerto y se procede a matar a la tarea. Si es una Morty, se mata a la tarea. En ambos casos, usamos la función killcurrent_task().

### 1.7.6. Modo debug

Las partes que operan con este modo se encuentran en varias partes del sistema.

En la interrupción del teclado, setea la variable modo_debug, encargada de indicar si esta activo o no.

En el manejo de las excepciones, pasamos la pila los parámetros a ser printeados en el cartel de debugueo. Utilizamos el ss y el esp pasados por la pila al comienzo de la excepción para obtener los últimos cinco valores del stack de nivel 3. También de la pila, obtenemos el cs y el eip de la tarea que produjo la excepción. Llamamos a la función activar_desactivar_excepcion que se encargará de poner en TRUE la variable 'excepcion', la cual indicará que se produjo una. Luego, chequeamos si está activo el modo debug. De ser así, procedemos a imprimir el cartel de debug con todos los datos pedidos. Una vez hecho esto, procedemos a matar la tarea que produjo la excepción para luego saltar a la tarea Idle. Una vez en la Idle, para evitar que se realice el cambio de contexto en la interrupción de clock, realizamos una serie de chequeos en la misma. Si la variable 'excepcion' esta en TRUE y esta el modo debug activado, quiere decir que se produjo una excepción, están printeados los registros en pantalla, se salto a la tarea Idle y todavía no fue presionada la tecla 'y' de vuelta. En ese caso, simplemente saltamos al final y realizamos el iret para volver a la tarea Idle. Si la variable 'excepcion' esta en TRUE pero el modo debug esta desactivado, quiere decir que hay que continuar con el normal funcionamiento del sistema. Seteamos 'excepcion' en FALSE para evitar que se interrumpa el sistema indebidamente si se activa el modo debug y realizamos el cambio de contexto normalmente. Finalmente, si está activo el modo debug pero la variable

'excepcion' está en FALSE, quiere decir que se presiono la tecla 'y' pero aún no se produjo ninguna excepción, por lo que seguimos con procedimiento usual de cambio de contexto.

### 1.7.7. Funciones Auxiliares

En este apartado se encuentran todas las funciones auxiliares que utilizamos para la implementación del tp.

- killcurrent_task: Mata la tarea actual indicada por la variable global current_task (se pone en 0 el campo is_alive en el arreglo de Scheduling, de modo que la función sched_nextTask ya no la considerará). Además, se verifica los siguientes casos:
  - Si la current_task corresponde a alguna Cronenberg y además esa tarea genero una excepción se procede a marcar dicha muerte en el tablero.
  - Si la current_task corresponde a alguna tarea Rick o Morty de cualquier universo, se procede a finalizar el juego y queda como ganador el universo opuesto.
  - Si la variable cronenbergs_dead es igual al total de tareas Cronenberg se procede a finalizar el juego quedando como ganador el que mas mentes conquisto o en un empate según corresponda.

  Una vez que se halla matado a la tarea, realizaremos un jmp far a la tarea Idle, de modo que esta misma ocupe el resto del tiempo hasta el próximo clock.
- reset_screen: Setea la pantalla con las posiciones actuales de cada tarea tarea viva y los portales que fueran creados actualmente.
- iniciar_pantalla:Esta función se encarga de iniciar la pantalla del juego, el tablero y printea la posición inicial de cada una de las tareas según corresponda.
- portalCreatReset: Setea en TRUE las variables globales portalNotYetCreated_byC137 y portalNotYetCreated_byD248 para indicar que los Rick correspondientes a sus universos pueden hacer uso del arma de portales. La llamaremos en cada tick de reloj.
- imprimir_registros: Se encarga de imprimir por pantalla todos los datos requeridos para el modo debug dentro de una tabla.
- print_exception: Printea en pantalla la excepción correspondiente recibida como parámetro.
- activar_modo_debug: Setea la variable global excepción según su estado actual.
- verificar_modo_debug: Verifica si esta activado el modo debug, retornando la variable global booleana que indica el estado actual.
- activar_desactivar_excepcion: Setea la variable global excepción según su estado actual.
- verificar_excepcion: Verifica si hay un excepción en curso,retornando la variable global para dicha acción.

- getRandomPosicion: La utilizamos para, dado un índice, obtener una posición ”aleatoria” (se utilizará el índice para obtener un par de números de un arreglo estático en memoria) en el cual posicionar inicialmente a una tarea.
- hacer_resto_positivo: La utilizamos para calcular el modulo de los desplazamientos dado que pueden ser negativos, toma 2 parámetros un entero x que representa el calculo del desplazamiento de cada coordenada y el divisor que representa el total de filas o columnas según la coordenada que corresponda, en caso de que el primer parámetro sea negativo se le suma el divisor.