DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming - Asimilación Cronenberg

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Venegas, David Alejandro	783/18	davidalevng@gmail.com
Bustamante, Luis Ricardo	43/18	luisbustamante097@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

0. Introducción

El objetivo de este Trabajo Práctico es desarrollar un sistema operativo aplicando los conocimiento adquiridos en las clases prácticas y teóricas de *System Programming* de la materia de organización del computador 2. El sistema consiste en un juego llamado *Asimilación Cronenberg* inspirado en la serie Rick and Morty que corresponde a un multiverso en 'Memoria'. El sistema mínimo desarrollado es capaz de correr hasta 24 tareas concurrentemente a nivel usuario, las cuales pueden vivir o morir de acuerdo a la dinámica del juego. Esta se basa en simular una guerra entre conjuntos de tareas: dos pares de *Rick and Morty* y las *Cronenberg*.

Inicialmente en el juego las tareas son ubicadas de forma aleatoria dentro de un mundo, donde una vez comenzado cada tarea puede vivir o morir, pero no revivir. El juego terminará cuando alguno de los Ricks tenga todas las mentes de otras tareas y el otro ninguna, o alguno de los Ricks o Mortys muera. Durante la simulación, los Ricks utilizan su arma de portales para moverse por el mundo o capturar mentes (tareas).

En la parte técnica, se utiliza lenguaje C y Assembly para el desarrollo y como entorno de pruebas el programa *Bochs* para simular una computadora IBM-PC, además de que este permite realizar tareas de debugging a muy bajo nivel. El presente trabajo se encuentra organizado por ejercicios que se resolvieron de forma gradual para completar el sistema.

1. Ejercicio I

1.1. GDT

Lo primero para empezar a desarrollar el juego en cuestión, es definir la Tabla de Descriptores Globales con los descriptores de segmentos necesarios. Se conoce que el primer descriptor es nulo, y por restricciones del trabajo práctico otros están reservados, así que los descriptores agregados empiezan en la posición 8 (contando desde cero). Primero se crean descriptores para código a nivel del kernel con nivel de protección 0, y nivel de usuario con de nivel de protección 3. Luego, se añaden descriptores de código de nivel de protección 0 y 3 para kernel y usuario respectivamente.

Los segmentos de datos y códigos direccionan los primeros 137MiB de memoria, utilizando bloques de 4KiB al setear el bit de granularidad en 1.

Ahora bien, se pasará a describir cada uno de los atributos de los segmentos 8 al 11 de la GDT:

- **Base:** debido a que se utiliza una segmentación *flat*, la base de cada registro se encontrara en el inicio de la memoria RAM, es decir en `0x00000000`.
- **Límite:** por el enunciado se sabe que se requiere que los segmentos sean de tamaño 137MiB, primero consideramos que el límite se toma de forma excluyente por lo que se debe restar en uno al valor obtenido. Y por otro lado, por lo que se especifica luego en el campo de Granularidad se debe calcular el límite en base a pequeños bloques de 4KiB, por lo que el límite se calcula mediante la siguiente formula: `(137MiB / 4KiB) - 1` = $(32071)_{10}$ = `0x88FF`.
- **Tipo:** para los descriptores 8 y 10 que representan a los segmentos de código, el tipo será **10** (Execute/Read), mientras que para los descriptores 9 y 11 que refieren a los segmentos de datos será **2** (Read/Write).
- **Descriptor type (S):** en los cuatros registros representan segmentos de código o data, por lo tanto este campo esta seteado a 1.
- **DPL**: El nivel de privilegio para cada segmento es seteado dependiendo si es para kernel o user, es decir, si requiere nivel kernel entonces DPL = `0x0` y si es nivel user DPL = `0x3`.
- **Granularidad (G):** La granularidad para los 4 descriptores será seteada a 1 porque como fue mencionado anteriormente, necesitamos acceder a direcciones superiores al MiB.
- **P, L, D/B, AVL:** Sus valores son: P=1 (porque los segmentos están disponibles para ser usados), L=0 y D/B=1 (porque se esta trabajando en 32 bits) y AVL=0.

En resumen, en la GDT vamos a tener lo siguiente:

Descripción	Index	Base	Limite	Tipo	S	P	DPL	AVL	L	D/B	G
nulo	0	0x0	0x0	0	0	0	0	0	0	0	0
reservado	1-7	0x0	0x0	0	0	0	0	0	0	0	0
GDT_IDX_CODE_0	8	0x0	0x88FF	10	1	1	0	0	0	1	1
GDT_IDX_DATA_0	9	0x0	0x88FF	2	1	1	0	0	0	1	1
GDT_IDX_CODE_3	10	0x0	0x88FF	10	1	1	3	0	0	1	1
GDT_IDX_DATA_3	11	0x0	0x88FF	2	1	1	3	0	0	1	1

Cuadro 1: GDT con los segmentos de código de nivel 0 y 3

1.2. Modo protegido y la pila del kernel

Lo primero que se debe hacer es pasar a modo protegido para habilitar `A20` debido a que los segmentos utilizan más de un 1 Mib. Esto se hace llamando a la subrutina `enable_A20` provista por la cátedra. Luego se carga la GDT por medio de la instrucción `LGDT`, para luego setear el primer bit del registro CR0 en 1 y finalmente realizar un "jump far" mediante la instrucción `jmp KERNEL_CS:protected_mode` donde `KERNEL_CS` tiene el valor de `0x0040`.

```
call A20_enable
call A20_check

lgdt [GDT_DESC]     ; Cargar la GDT

mov eax, cr0  ; Setear el bit PE del registro CR0
or eax, 0x1
mov cr0, eax

jmp KERNEL_CS:protected_mode     ; Saltar a modo protegido

BITS 32
protected_mode:
```

Ahora ya en modo protegido se inicializan todos los selectores de segmentos al valor de `KERNEL_DS`, es decir `0x0050`. Y por último se define la pila del kernel mediante `mov ebp, 0x27000` y `mov esp, ebp`.

1.3. Segmento de Video en la GDT

Adicionalmente, se completa la GDT creando un descriptor de segmento para vídeo con base en la dirección `0xB8000`, DPL=0 y Tipo=2 para direccionar 8000 bytes ($80 * 50 * 2$) correspondientes a la pantalla. Cabe destacar que se utiliza G=0 porque no necesitamos direccionar a más de 1 Mib. Y debido a esto último sabemos que el valor del límite es igual a `(8KiB -1)` = $(7999)_{10}$ = `0x1F3F`. El resto de campos se setearán como los anteriores descriptores.

Descripcion	Index	Base	Limite	Tipo	S	P	DPL	AVL	L	D/B	G
nulo	0	0x0	0x0	0	0	0	0	0	0	0	0
reservado	1-7	0x0	0x0	0	0	0	0	0	0	0	0
GDT_IDX_CODE_0	8	0x0	0x88FF	10	1	1	0	0	0	1	1
GDT_IDX_DATA_0	9	0x0	0x88FF	10	1	1	0	0	0	1	1
GDT_IDX_CODE_3	10	0x0	0x88FF	2	1	1	3	0	0	1	1
GDT_IDX_DATA_3	11	0x0	0x88FF	2	1	1	3	0	0	1	1
GDT_IDX_VIDEO	12	0xB8000	0x1F40	2	1	1	0	0	0	1	0

Cuadro 2: GDT con el segmento de vídeo agregado en el índice 12.

1.4. Inicializar la pantalla con subrutina en ASM

En esta parte, para inicializar pantalla se utiliza el segmento declarado en el punto anterior.

```
mov ax, KERNEL_VIDEO    ;Muevo a AX un selector para segmento 12 (pantalla)
mov fs, ax              ;Uso fx para el selector de la memoria de video
call limpiar_pantalla
call pintar_pantalla
```

Entonces, usando `FS` el selector de segmento correspondiente al vídeo se itera sobre los píxeles de la pantalla, primero para limpiarla (pintando todo de negro) y luego para pintar el área del tablero de juego teniendo en cuenta los offsets correspondientes.

2. Ejercicio II

2.1. IDT

En esta sección para el manejo de interrupciones se completa una tabla IDT que permite ejecutar la rutina de atención correspondiente para cada evento. Esta tabla cuenta con un selector de segmento, offset, y bits de nivel protección y tipo de entrada.

La IDT se representa mediante un arreglo (de tamaño 255) de `idt_entry`, esta se inicializa en el kernel llamando la función `idt_init()`. Las entradas de la tabla son creadas mediante un macro que define la función en ASM `_isrX` donde *X* es el número de interrupción para ser atendida.

Existen 3 tipos de entrada para la arquitectura Intel, pero para el presente trabajo se utilizarán solo interrupciones del tipo `Interrupt Gate`, las cuales desactivan las interrupciones al iniciar la rutina de atención. Por ahora, aún muchas de estas rutinas no están definidas (se les contrarresta con un loop infinito), pero serán completadas más adelante.

Se completan en la IDT con los siguientes valores:

- **Offset:** se utiliza la dirección de memoria del código de la interrupción.
- **Selector de segmento:** se toma el índice de selector de segmento de código del kernel nivel 0 en la GDT shifteado 3 bits a la izquierda, es decir, `0040h`.
- **Atributos:** segmento presente con $P = 1$, nivel de privilegio 0 en DPL, y el tamaño del gate es de 32 bits con $D = 1$. En definitiva `0x8E00`.

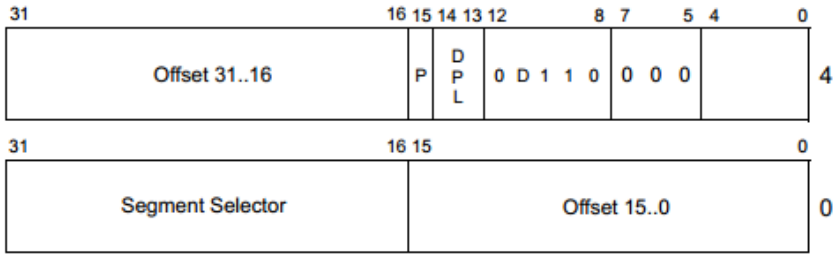

Figura 1: Interrupt Gate Descriptor para la IDT

2.2. Integrando la IDT en el Kernel

Luego de llenar las entradas de la IDT en el archivo `idt.c`, como se menciona anteriormente en el kernel se llama la función para inicializar la tabla con `call idt_init` y finalmente se utiliza la instrucción `lidt [IDT_DESC]` para cargar la dirección y el tamaño de la IDT en el registro IDTR.

Además, se hace la prueba de que las interrupciones sean correctamente atendidas mediante una división por 0, la cual deberá activar la excepción *#DE* (Divide Error Exception), la cual es visualizada por la pantalla de Bochs como un mensaje diciendo: `Excepcion 0`.

3. Ejercicio III

3.1. Extensión de la IDT

Debido a que las interrupciones producidas por el sistema (excepciones) ya pueden ser atendidas por lo trabajado en el ejercicio anterior, se procede a agregar 2 nuevas interrupciones que son producidas por Hardware, la interrupción por clock y la interrupciones producidas por el teclado. Ambas son atendidas por una rutina (ISR) la cual tendrá un comportamiento específico reflejado directamente en la pantalla del sistema. Los descriptores de ambos tienen los mismos atributos y el mismo selector de segmentos que los descriptores de las excepciones tratadas anteriormente, y los índices de cada uno son el 32 y 33 respectivamente. Por último se agregarán 3 descriptores de interrupción los cuales serán completados en los próximos ejercicios y estarán mapeadas a los índices 137, 138 y 139.

3.2. Rutina de interrupción: Clock

La interrupción por clock es de vital importancia dado que está permite implementar un scheduler para tener un sistema concurrente, pero esto se verá al final del informe. Fuera de eso, en un pasado esta era la forma de medir el tiempo transcurrido, pero lamentablemente Bochs no emula correctamente esto, por lo que (por lo pronto) va a ser utilizado fundamentalmente para saber que el sistema está corriendo normalmente, lo cual se va a ver mediante un puntero en la esquina inferior derecha que va a estar girando continuamente. Esto es logrado mediante la siguiente rutina de atención:

```
_isr32:
    pushad
    call pic_finish1
    call nextClock
    popad
    iret
```

Donde primero se mantiene la integridad de todos los registros de propósito general, luego se le indica al *PIC* que se va a atender la interrupción entrante mediante la función `pic_finish1`. Cabe destacar que no es fundamental que esta instrucción se lleve a cabo al principio ya que es solo para avisarle al *PIC* que acaba de ser atendido. A continuación, se llama a la función implementada por la cátedra que dibuja el puntero en el giro que le corresponda, y por último se retorna al sistema en el estado antes de haber atendido la interrupción.

3.3. Rutina de interrupción: Teclado

La rutina de atención del teclado es otra rutina bastante interesante ya que se va a disparar cada que una tecla sea pulsada, y recibirá el *Scan Code* del carácter presionado en cuestión. Por motivos del enunciado, solo se limitará a recibir las teclas de números del 1 al 9. Es necesario aclarar que en realidad se obtendrá el *Make Code* del dispositivo de entrada, ya que este hace referencia al valor emitido al presionarse la tecla (y no al soltarse). Además, el dispositivo que controla la entrada del teclado mantendrá un buffer de las teclas presionadas hasta que el sistema reciba y resuelva cada una de las interrupciones del teclado.

```
_isr33:
    pushad
    in al, 0x60
    push eax
    call printScanCode
    add esp, 4
    call pic_finish1
    popad
    iret
```

Para determinar el valor a imprimir se usa la instrucción `in AL, 0x60` que toma el valor de puerto Entrada/Salida `0x60` y lo copia en el registro `AL`, luego pasa este valor por pila a la función `printScanCode` e imprime el valor correspondiente en la posición esperada. El comportamiento de dicha función esta expresado en el siguiente cuadro, donde primero se verifica que el valor recibido no es un *Break Code*, y luego se chequea que las teclas pulsadas sean las teclas numéricas del 1 al 9. Se puede apreciar que el *Scan Code* está corrido en una unidad porque el código emitido por la pulsación de cada tecla no corresponde al código ASCII ni al orden correspondiente a los números, si no que se corresponde a una configuración vieja de los teclados.

```
void printScanCode(uint8_t scanCode)
    if!(scanCode & 0x80))
        if ((0x2 <= scanCode) & (scanCode <= 0xA) )
            print_hex(scanCode-1,2,80-2,0, C_BG_BLACK| C_FG_WHITE);
```

Por último se utiliza la función de la cátedra `print_hex` para escribir en la esquina superior derecha la tecla presionada en cuestión.

3.4. Rutina de interrupción: 137,138 y 139

Las rutinas asociadas a interrupciones de software o llamados al sistema (syscalls) se reservan en las entradas 137, 138 y 139. Para esto se agrega un nuevo macro `IDT_ENTRY_SW(número)` con un nivel de privilegio 3, es decir, el atributo DPL de la entrada en la IDT será 3. Más adelante se reemplazará el código de estas instrucciones pero por el momento estas sólo modifican el valor de `eax` por `0x42`, `0x43` y `0x44` respectivamente.

```
_isr137:
    mov eax, 0x42
    iret
```

```
_isr138:
    mov eax, 0x43
    iret
```

```
_isr139:
    mov eax, 0x44
    iret
```

4. Ejercicio IV

4.1. Paginación

Para manejar la memoria del sistema operativo desarrollado en este trabajo práctico se utiliza el esquema de paginación con dos niveles de protección. En este la memoria se encuentra dividida en bloques fijos de 4 KiB, lo cual facilita su uso y organización; sin embargo, trabajar con procesadores Intel requiere mantener el sistema de segmentación flat, en este caso de 137Mib para cada segmento. Además, existen diversos tipos de paginación soportados por el procesador, aunque para el presente trabajo se va a limitar a emplear paginación de 4Kib sin PAE (Page Address Extension).

En este esquema se utiliza el registro CR3 para acceder a la base de un directorio de páginas (*PD* por sus siglas en inglés) compuesto por entradas (PDE) apuntando a la base de un tabla de páginas. Estas tablas contienen entradas (PTE) que apuntan a la base de una pagina de 4 Kib (memoria física), y finalmente se direcciona la posición exacta gracias al offset, como se puede apreciar en la Figura 2 obtenida del manual de Intel.

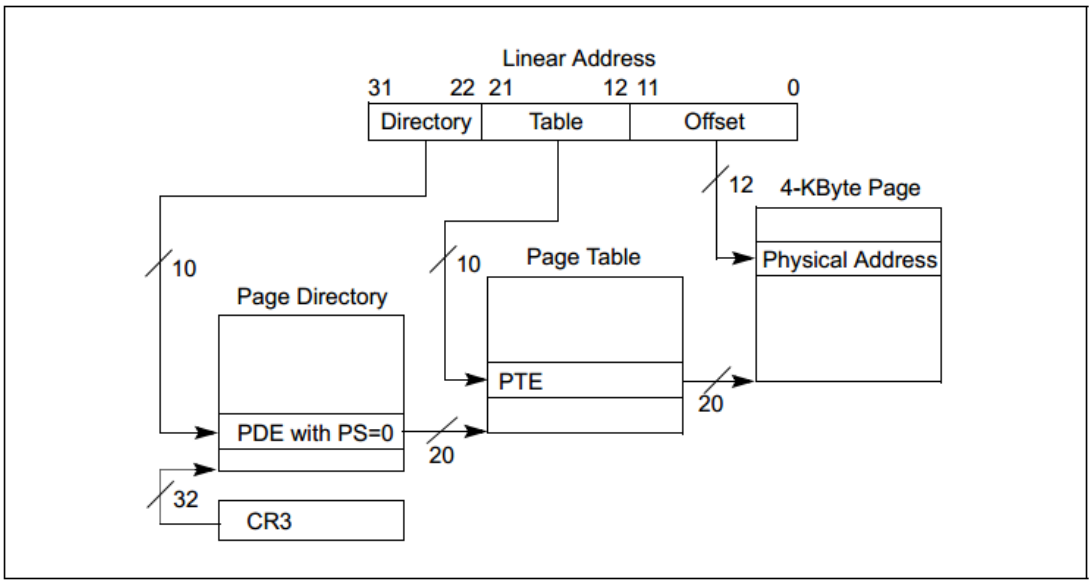

Figura 2: Traducción de una dirección lógica a una pagina de 4Kib

A continuación se describen los valores de la CR3, PDE y PTE:

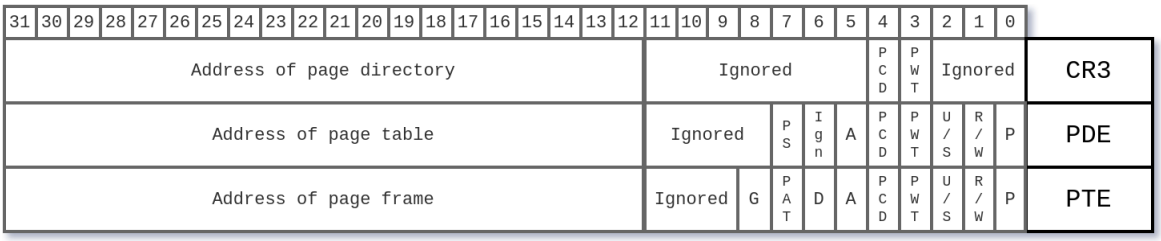

Figura 3: Representación de los atributos del CR3, de los PDE y los PTE.

CR3

- **Base:** dirección base del directorio de páginas (PD). Bits 31 a 12.
- **PWT (Page-level write-through):** atributos referentes al manejo de la cache. Bit 3.
- **PCD (Page-level cache disable):** atributos referentes al manejo de la cache. Bit 4.
- **Ignored:** bits ignorados que deben estar seteados en 0.

Page Directory Entry

- **Base:** dirección base de la tabla de páginas (PT). Bits 31 a 12.
- **P (Present):** si es 1 indica que la tabla está disponible. Bit 0.
- **R/W (Read/Write):** si es 1 indica que la tabla es escribible. Bit 1.
- **U/S (User/Supervisor):** indica el nivel de protección de la tabla. Bit 2.
- **PWT (Page-level write-through):** atributo referente al manejo de la cache. Bit 3.
- **PCD (Page-level cache disable):** atributo referente al manejo de la cache. Bit 4.
- **A (Accessed):** si es 1 indica si fue accedido. Bit 5.
- **PS (Page Size):** si es 0 indica que las páginas tienen un tamaño de 4KiB. Bit 7.
- **Ignored:** bits ignorados que deben estar seteados en 0.

Page Table Entry

- **Base:** dirección base de la tabla de páginas (PT). Bits 31 a 12.
- **P (Present):** si es 1 indica que la tabla esta disponible. Bit 0.
- **R/W (Read/Write):** si es 1 indica que la tabla es escribible. Bit 1.
- **U/S (User/Supervisor):** indica el nivel de protección de la tabla. Bit 2.
- **PWT (Page-level write-through):** atributo referente al manejo de la cache. Bit 3.
- **PCD (Page-level cache disable):** atributo referente al manejo de la cache. Bit 4.
- **A (Accessed):** si es 1 indica si fue accedido. Bit 5.
- **D (Dirty):** si es 1 indica si la memoria referenciada por esta tabla fue modificada. Bit 6.
- **PAT (Page-attribute table):** atributo referente al manejo de la cache. Bit 7.
- **G (Global):** atributo referente al manejo de la cache. Bit 8.
- **Ignored:** bits ignorados que deben estar seteados en 0.

La memoria del kernel se organiza mediante *identity mapping* donde las direcciones virtuales son iguales a las físicas. Cabe mencionar que las direcciones virtuales son las direcciones que el programador ve, en los manuales de Intel son llamadas direcciones lineales pero en la totalidad de éste informe serán referidas como virtuales. Para inicializar paginación primero se debe completar el directorio de paginas del kernel para que mapee las direcciones desde `0x00000000` hasta `0x003FFFFF`, es decir los primeros 4 MiB, para luego inicializar dicho directorio en la dirección `0x27000` y la tabla de páginas en la `0x28000`. Dado que con una página se pueden direccionar hasta 4Mib, con una sola es suficiente.

4.2. Activar Paginación

Ahora para que el procesador reconozca que paginación fue activada se debe modificar el bit menos significativo del registro de control `cr0`.

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

4.3. Rutina para imprimir el número de libreta

Ahora para probar que todo lo anteriormente descripto funciona correctamente, en esta sección se usa el macro `imprimir_texto_mp` provisto por la cátedra. Esto permite detectar si hubo algún error al implementar el mecanismo de paginación, en este caso se generaría una excepción del tipo `#PF`.

```
print_text_pm msj_libreta, msj_libreta_len, 0x07, 49, 0
```

5. Ejercicio V

5.1. Administración de memoria

En esta sección se escribe la rutina `mmu_init` encargada de inicializar las estructuras necesarias para administrar la memoria en el área libre de kernel. Las estructuras utilizadas son dos contadores de paginas libres,de los cuales uno se utiliza para indicar el del kernel inicializado en `0x100000` (direcciones según la cátedra). Este valor es la dirección física donde empiezan las paginas libres para cada caso, y cada vez que se solicite una nueva página libre, se utiliza este valor para que luego se le sume `0x1000` o mejor dicho 4 KiB, que es el tamaño de una página, para apuntar así a la siguiente página libre. Este manejo de memoria es volátil dado que se asignará memoria a quien la pida hasta que la misma colapse, pero a efectos prácticos nos servirá para poder simular una *MMU*.

```
void mmu_init() {
  next_page_libre_kernel =       BASE_KERNEL_FREE_AREA;
}
uint32_t mmu_nextFreeKernelPage() {
  int nextPage = next_page_libre_kernel;
  next_page_libre_kernel += PAGE_SIZE;
  return nextPage;
}
```

5.2. Mapeo de páginas

Ahora se definen las rutinas encargadas de mapear y desmapear las páginas de memoria: `mmu_mapPage`y `mmu_unmapPage` respectivamente. Estas funciones tendrán la capacidad de brindar una página de 4 KiB a cualquier proceso que lo pida, y luego poder desmapear la página solicitada. La acción de desmapeo solo libera la memoria utilizada pero no la dejará disponible para ser usada luego.

Para la función `void mmu_mapPage(uint32_t cr3, uint32_t virtual, uint32_t phy, uint32_t attrs)` se procederá a seguir este pseudocódigo para finalmente obtener la pagina de memoria solicitada.

1. Dividir la dirección virtual a mapear en directoryIdx, tableIdx y offset.
2. Usando el parámetro `CR3`, calcular la dirección de la PDE.
3. Revisar si el bit presente de la PDE es 0. De ser así pedir una nueva página para la page table, completarla con ceros y completar la PDE.
4. Obtener la page table de la PDE.
5. Usando el puntero al comienzo de la page table y el campo tableIdx obtener la PTE.
6. Completar la PTE con el marco de página que se busca mapear.
7. Completar los atributos en la PDE y PTE con los atributos de entrada de la función.
8. Llamar a la función `tlbflush()` para invalidar todas las entradas del búfer de traducción anticipada (o TLB por sus siglas en inglés).

Es interesante destacar que en un sistema operativo real sería muy ineficiente llamar `tlbflush()` cada vez que se va a mapear una dirección de memoria, sin embargo en este trabajo a efectos prácticos se hace para evitar que se generen errores con la TLB (que puede contener información desactualizada).

De forma análoga para la función `uint32_t mmu_unmapPage(uint32_t cr3, uint32_t virtual)` se hace una suerte del proceso inverso:

1. Dividir la dirección virtual a mapear en directoryIdx, tableIdx.
2. Obtener la dirección de la PDE correspondiente.

3. Si el bit presente de la PDE es 0, no es necesario hacer nada.
4. Si es igual a 1 se debe invalidar toda la page table correspondiente con ceros en cada entrada.
5. Llamar a la función `tlbflush()`.

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

En esta sección se escribe la rutina `mmu_initTaskDir` encargada de inicializar un directorio de páginas y tablas de páginas válido para una tarea de acuerdo al mapa de memoria provisto por la cátedra en la figura 4. Esta rutina mapea las páginas del mundo Cronenberg donde se ubica cada tarea a partir de la dirección virtual `0x08000000` (128MB). Esta función recibe la dirección física donde se encuentra la tarea en cuestión y una dirección física en la que se tendrá que copiar dicha tarea. Cabe recordar que cada tarea ocupa dos páginas dado que tendrá los primeros 4 KiB para el código en sí, y el espacio restante será para el stack del mismo.

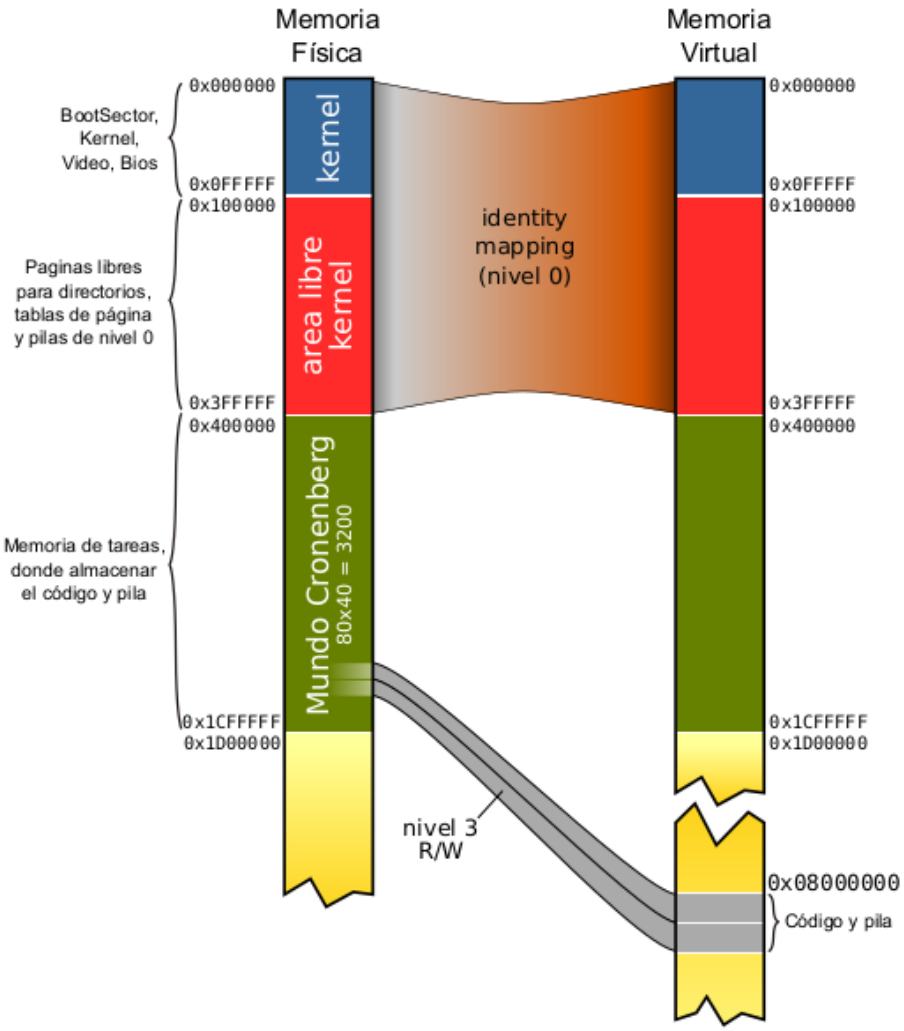

Figura 4: Mapa de Memoria provisto por la cátedra

Para esta parte es importante recordar que todas las tareas del sistema van a tener un diferente Page Directory ya que todas van a tener diferente código mapeado en diferentes lugares. En este esquema se mapean los primeros 4 MiB con *identity mapping* correspondientes al kernel, ya que ante cualquier evento del sistema el área del kernel debe ser accesible por el mismo kernel, como por ejemplo puede ocurrir durante una interrupción; las rutinas de atención de interrupciones deben ser alcanzables durante la ejecución de tal tarea.

Una vez cumplido este requisito se procede a copiar la tarea en cuestión en el destino solicitado. Es necesario aclarar que la dirección obtenida sera una dirección "aleatoria" cuyo origen y comportamiento serán especificados luego. Por lo pronto esta será la dirección dentro del mundo Cronemberg donde irá la tarea, y esta tarea será mapeada a la dirección virtual `0x08000000`.

Una mejor especificación de los pasos para construir el esquema de paginación de una tarea son tenidos en el siguiente pseudocódigo:

1. Solicitar una página libre para el PD.
2. Solicitar una página libre para el PT.
3. Construir un esquema de paginación con *Identity Mapping* para los primeros 4MB (como se mencionó anteriormente).
4. Identificar el código de la tarea que debe ser copiado desde el kernel (src)(recibido como parámetro).
5. Identificar la posición de memoria donde copiar el código (dst)(recibido como parámetro).
6. Mapear de ser necesario la posición destino o fuente del código. (dado que a priori no sabemos si las posiciones están mapeadas).
7. Copiar las dos paginas de la tarea.
8. Mapear la tarea copiada, al nuevo esquema de paginación que se está construyendo.
9. Desmapear de ser necesario las páginas mapeadas para poder copiar.
10. Retornar el Page Directory nuevo para que la tarea lo use como `CR3`.

5.4. Switch Kernel-Task

Ahora bien, para probar las rutinas desarrolladas se construye un mapa de memoria para tareas y se intercambia con el del kernel, para luego cambiar el color de fondo del primer carácter en pantalla. Esto se hace para verificar el correcto funcionamiento de memoria, dado que de no ser así obtendremos una excepción del tipo `#PF`. La idea de esta función es simular que hay una tarea en una dirección del área libre del kernel llamando a `mmu_initTaskDir`, la cual deberá ser mapeada al principio del Mundo Cronenberg, por lo que luego de llamar a dicha función se hará el cambio de `CR3` a la dirección obtenida del nuevo Page Directory.

Este código es solo de prueba y queda comentado en la versión final:

```
;signatura de la función:
;   uint32_t mmu_initTaskDir(uint32_t taskDir, uint32_t taskPage)
push 0x400000       ;uint32_t taskPage
push 0x300000       ;uint32_t taskDir   OBS:dir random del area libre del kernel

call mmu_initTaskDir
mov cr3, eax

print_text_pm msj_libreta, 1, 0x44, 0, 0
```

6. Ejercicio VI

6.1. Tareas y la GDT

Para utilizar tareas, el procesador de Intel provee ciertas estructuras e instrucciones que facilitan su uso a nivel del sistema operativo. La TSS es un segmento que permite almacenar toda la información relativa a la ejecución de una tarea, particularmente, su estado o *contexto*, esta actúa como una fotografía del estado actual del sistema en un cierto punto que permite guardar y restaurar información durante el cambio de tareas. De igual forma que los otros segmentos, cada TSS debe ir a una entrada de la GDT.

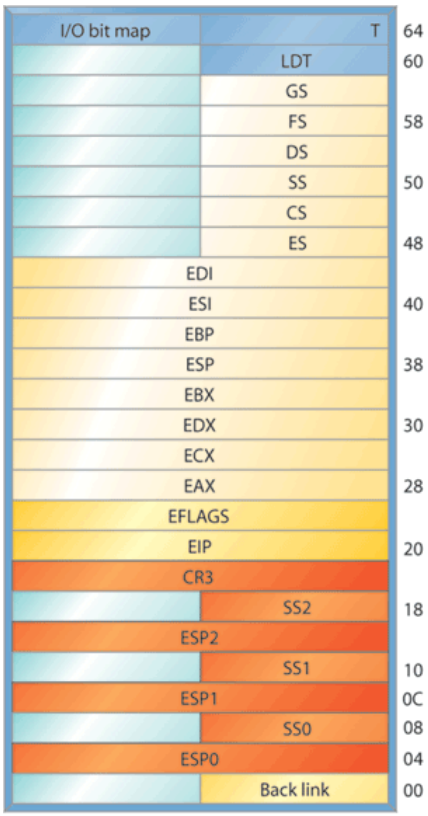

Figura 5: Representacion de la TSS

Primero, en la GDT utilizando la función `tss_create_gdt_entry` se deben agregar las entradas correspondientes para ser usadas como descriptores de segmento en la TSS. Mínimamente, para esta sección se inicia con la tarea Inicial y la tarea Idle, de esta forma se crea un arreglo de *TSS* (estructura definida por nosotros) tal que `tss tss_array[24]` permita acceder a los diferentes segmentos de tarea, en especial por ahora a `tss_initial` y `tss tss_idle`.

Para la tarea inicial solo es necesario hacer:

```
tss_create_gdt_entry(GDT_IDX_TASK_INIT, (uint32_t) &tss_initial, 0b00);
```

6.2. TSS y Tarea Idle

Luego, para la tarea idle se usa la función `tss_init_idle()` y se va a inicializar en 0x0 la mayoría de sus posibles valores, destacando los más relevantes:

- **esp0**: 0x0 ya que no va a haber un cambio de privilegios.
- **ss0**: 0x0 idem.

- **cr3**: `KERNEL_PAGE_DIR`, es decir, la dirección del page directory del kernel.
- **eip**: `TASK_IDDLE_CODE_ADDR` en la dirección de comienzo de la tarea idle.
- **eflags**: 0x00000202 con la interrupciones encendidas.
- **esp**: 0x00027000 en el tope del stack del kernel.
- **ebp**: 0x00027000 en la base del stack del kernel.
- **es**: `GDT_SEG_SEL_DATA_0` es decir, los selectores de segmento del kernel.
- **cs**: `GDT_SEG_SEL_CODE_0` selector de segmento para la entrada de código del kernel de la gdt.
- **ss**: `GDT_SEG_SEL_DATA_0`.
- **ds**: `GDT_SEG_SEL_DATA_0`.
- **fs**: `GDT_SEG_SEL_DATA_0`.
- **gs**: `GDT_SEG_SEL_DATA_0`.
- **iomap**: 0xFFFF valor necesario para no utilizar IOMAP.

6.3. Tarea Inicial - Entrada en al GDT

Los descriptores de TSS permiten asociar los segmentos a un entrada de la GDT con la siguiente estructura:

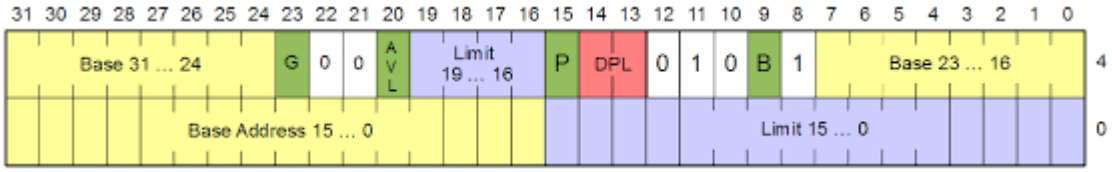

Figura 6: Descriptor de la TSS

Para agregar las entradas a la GDT como se mencionó anteriormente, se utilizó la siguiente función:

```
void tss_create_gdt_entry(uint32_t index, uint32_t base, uint8_t dpl){
    gdt[index].limit_0_15   = sizeof(tss)-1;        // minimo para una TSS
    gdt[index].limit_16_19  = 0x0;
    gdt[index].base_0_15    = base & 0xFFFF;
    gdt[index].base_23_16   = (base >> 16) & 0xFF;
    gdt[index].base_31_24   = (base >> 24);
    gdt[index].type         = 0b1001;               // donde B = Busy == 0
    gdt[index].s            = 0;                    // descriptor de Sistema
    gdt[index].dpl          = dpl;
    gdt[index].p            = 1;
    gdt[index].avl          = 0;
    gdt[index].l            = 0;
    gdt[index].db           = 1;
    gdt[index].g            = 0;
}
```

Como el mecanismo de manejo de tareas en el procesador nos obliga a realizar un seteo “simbólico” de las estructuras de una tarea antes de empezar y se utliza `tss_init` para lo mencionado. Cargamos la tarea inicial:

```
    ;   GDT_IDX_TASK_INIT = 13 = 0xD ; TI=0 ; RPL = 00
    ;        => selector = (0xD<<3) || 000 = 0x68
    mov ax, 0x68
    ltr ax
```

6.4. Tarea Idle - Entrada en al GDT

En esta parte se carga la entrada de la tarea Iddle en la GDT de la misma forma que se hizo con la inicial, pero utlizando la función `tss_init_idle`. Y a partir de ahora ya está lista para ser utilizadas más adelante en el kernel, tanto para iniciar el scheduler como desalojar tareas cuando ocurra una interrupción en el modo debugger.

```
; Saltar a la primera tarea: Idle
    ;   GDT_IDX_TASK_IDLE = 14 = 0xE ; TI=0 ; RPL = 00
    ;        => selector = (0xE<<3) || 000 = 0x70
    jmp 0x0070:0x0
```

Finalmente, se agregan las entradas faltantes de la TSS para la GDT llamando a `tss_init_others`

6.5. Constructor TSS

En esta sección se crea la rutina `new_task` que permite completar una TSS con los datos correspondientes (crea una tarea), y contruye el mapa de memoria adecuado mediante `mmu_initTaskDir`. Entrando en detalle, lo primero que se debe hacer es pedir una página libre del kernel para la pila de nivel 0, y luego con el índice pasado como parámetro acceder al array donde se encuentran las TSS almacenadas en memoria. Luego, obtenemos la dirección de la página física correspondiente al mundo cronenberg, en la cual se copiará el código de la task según el array de coordenadas aleatorias propuestas por la cátedra. Ya con esta información se puede mapear memoria e inicializar las TSS. La mayoria de los valores son inicializados en cero, sin embargo, los más resaltantes de esta configuración se detallan a continuación:

- **esp0**: `dir_pila_lvl_0` para acceder a la pila del kernel
- **ss0**: `GDT_SEG_SEL_DATA_0`, es decir, el selector de segmento de data del kernel de la GDT.
- **cr3**: `task_cr3` con el nuevo PD.
- **eip**: `TASK_CODE` que tiene la dirección del comienzo de la tarea en cuestión.
- **eflags**: `EFLAGS_INT_ON` con la interrupciones encendidas.
- **esp**: `TASK_CODE + PAGE_SIZE*2` que sería la ubicación del tope del stack de nivel 3
- **ebp**:`TASK_CODE + PAGE_SIZE*2` idem para la base.
- **es**: `GDT_SEG_SEL_DATA_3` es decir, los selectores de segmento de nivel 3.
- **cs**: `GDT_SEG_SEL_CODE_3` selector de segmento para la entrada de codigo de nivel 3.
- **ss**: `GDT_SEG_SEL_DATA_3`.
- **ds**: `GDT_SEG_SEL_DATA_3`.
- **fs**: `GDT_SEG_SEL_DATA_3`.
- **gs**: `GDT_SEG_SEL_DATA_3`.
- **iomap**: 0xFFFF.

7. Ejercicio VII

Esta es la parte mas integral de todo el trabajo, dado que es en este punto donde unimos cada una de las piezas que fuimos armando en cada uno de los ejercicios anteriores. Cabe destacar que fue la parte más compleja y más importante de todas, dado que fue aquí donde se empezaron a testear cada uno de los puntos anteriores con mucho cuidado. Dada la extensa cantidad de cosas plasmadas en código, cabe resaltar que algunas cosas no podrán ser alcanzadas por el informe debido a la complejidad de las implementaciones y se dejan en el código acompañadas de comentarios.

7.1. El Scheduler

En primer lugar uno de los pilares más grandes del kernel es el *scheduler*, que a pesar de ser uno muy rudimentario, ya que se utiliza un método *round-robin* de scheduling, resulta ser muy conveniente para la ejecución de las tareas. Éste en particular permite que el kernel opere con 24 tareas de forma concurrente, dando una sensación de simultaneidad entre los procesos como se puede apreciar en la Figura 7.

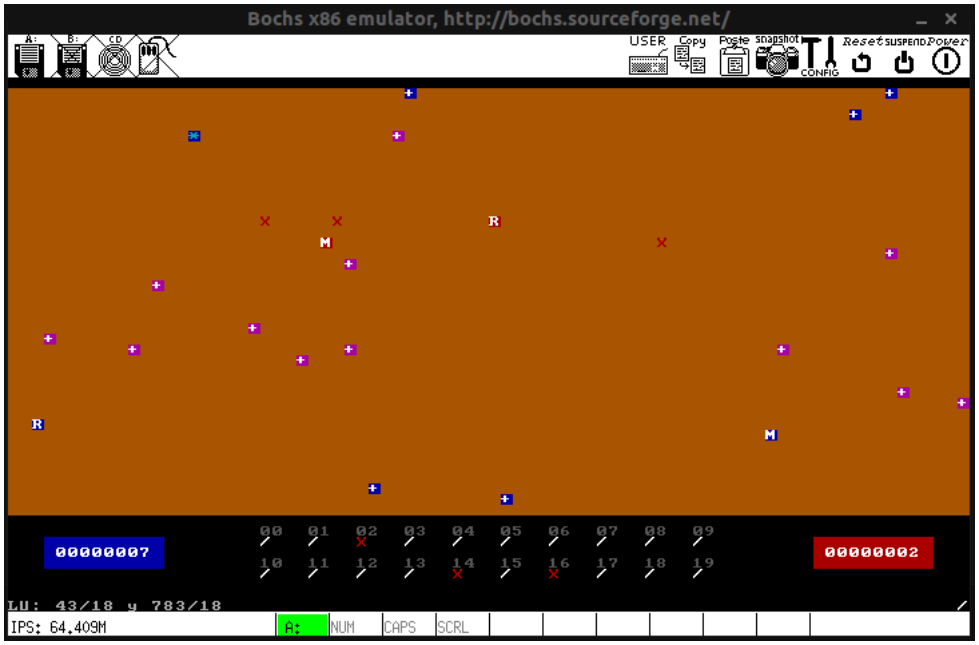

Figura 7: Captura del emulador corriendo el kernel.

El scheduler esta parado sobre dos estructuras fundamentales que son un arreglo de booleanos llamado `alive_tasks[24]` y un valor `actual_task`. El funcionamiento de ellos reside en mantener actualizado el estado de todas las tareas durante la ejecución del juego, dado que la política del scheduler es: “si una tarea esta muerta no la ejecutaré”. El criterio de muerte estará dado por las descriptas en el enunciado y también por las que nosotros consideremos. Por último se setean las variables necesarias para mantener la lógica del juego, estas son:

- `posTask_actual[24]`: mantiene la posición de todas las tareas del juego actualizadas.
- `posPortal_actual[2]`: mantiene la posición de los portales del juego actualizados. Estos portales solo podrán ser creados por la `SYSCALL usePortalGun` para un universo correspondiente, por lo que solo habrán dos portales como máximo la vez.
- `score[2]`: mantiene el puntaje de cada uno de los universos.
- `type[24]`: arreglo usado para saber el tipo de la tarea correspondiente.
- `contadorMorty[2]`: usado para que un Morty solo pueda usar el arma de portales cada 10 veces que la haya usado Rick.

El scheduler a pesar de tener una política muy básica, es completamente efectivo para hacer una ejecución competente de los recursos de la CPU. Ahora bien, en la Sección 7.5 se hablará de como entra en juego el scheduler para resolver los conflictos que pueden producir las tareas durante este proceso.

7.2. Cambio de Tarea

Esta acción esta representada por la función `sched_nextTask` la cual, mediante el resultado obtenido (el cual es el índice a la entrada correspondiente de la tarea a la GDT), hace respetar la política del scheduler. Todo esto ocurrirá durante la rutina de atención del clock, la cual cada cierta cantidad de tiempo (llamado *quantum*) pedirá al scheduler cual es la siguiente tarea a ejecutar, mediante la función antes descripta. El cambio de tarea también estará acompañado de la actualización del mapa, los puntajes y los relojes de cada tarea. Además, será acompañado de una función que chequeará si el juego se ha terminado basado en los parámetros proporcionados por la cátedra. En caso de que no haya ninguna tarea para ejecutar, pasará a ser ejecutada la tarea Idle, inicializada en la Sección 6.2.

7.3. Funcionamiento del Juego

La funcionalidad del juego es bastante sencilla, serán puestas 20 tareas Cronenberg (Todas compartiendo el mismo código), 2 tareas Rick y 2 tareas Morty, las cuales serán inicializadas con la función descripta en la Sección en la función `game_init`. En el Cuadro 3 tenemos por un lado los índices necesarios para la entrada de la GDT, y por otro lado el índice a la correspondiente tarea en para los arreglos mencionados en la sección anterior.

Índice	Índice de GDT	Tarea
0	15	Rick C-137
1	16	Morty C-137
2	17	Rick D-248
3	18	Morty D-248
4	19	Cronenberg-1
...	...	...
23	39	Cronenberg-20

Cuadro 3: Indices usados para la GDT

Por ultimo cabe destacar que para los arrays de dos posiciones (como `contador_morty` por ejemplo) se utiliza la primer posicion para representar al universo **C137** y la segunda posición para el universo **D248**.

7.4. Rutinas de Interrupciones

7.4.1. Syscall 137: UsePortalGun

Esta es la tarea mas importante del juego y permite desplazar a los Ricks y Mortys así como controlar mentes y escribir memoria. En el caso de los Ricks hay dos opciones importantes *cross* y *with_Morty*, esta última si es seleccionada se usa para mover a Morty a la par de Rick, sino solo a el y no tiene efecto al ser empleada por un Morty. En el caso de cross igual a cero el personaje usándola no se moverá sino que creara un portal que mapeara memoria a la dirección virtual 0x8002000 y permitirá escribir el código de otras tareas(controlar mentes).

Para el caso de *cross* igual a 1, tenemos que quien use el arma se desplazara a las coordenadas relativas que se proporcionen al ser llamada la syscall. Este caso tiene muchos posibles comportamientos, y se irán detallando uno por uno.

El primer caso es si un Rick quiere usar la pistola, pero sin morty, o bien Morty quiere usar la pistola para solo teletransportarse el. En este caso lo que se hará para cualquiera de los dos sera: primero guardar la dirección en la que se encuentra quien este por teletransportarse, luego desmapear la memoria que

tienen actualmente asignada, dado que se estarán moviendo a una nueva posición en memoria en la que tendrán que mapear la nueva dirección destino. Para que esto sea efectuado correctamente deben verificar si en la posición destino existe alguna tarea. En caso de que no haya ninguna tarea podrán copiar su código en la posición destino y se actualizara su nueva posición en la arreglo correspondiente. En el caso de que si haya una tarea se procederá a matar dicha tarea, para que esta sea desalojada del scheduler, y se actualizaran todas las variables necesarias para mantener la lógica del juego. Luego de esto se procederá de la misma forma descripta anteriormente.

Por otro lado, si un Rick quiere teletransportarse junto a su Morty, se debe realizar una búsqueda del Page Directory del Morty en cuestión, este puede ser encontrado en su TSS. Luego se procederá a realizar los pasos antes mencionados, hasta el punto en donde se tiene que hacer la copia del código en la posición destino ya que se necesita realizar un cambio temporal del Page Directory actual para hacer el correspondiente traspaso. Por ultimo se restablece el Page directory al adecuado y se termina la función.

Para el caso en que un Morty quiera saltar con el flag *withMorty* activado, el comportamiento esta indefinido. Por otra parte si se trata de un Cronenberg que trata de usar la portal gun, esto generará una excepción que hará que el sistema operativo mate la tarea (destáquese no el syscall).

7.4.2. Syscall 138: IamRick

Esta rutina puede ser invocada tanto por Ricks y Mortys como por Cronenbergs, e indican cuando una mente ha sido controlada. Esto en el juego se implementa como una función que dependiendo del código pasado: `C137` o `D248` actualiza el valor de arrays que permiten llevan la cuenta en cada ciclo de Clock del puntaje. Así cuando un Rick o Morty toma una mente, escribe código en la tarea haciendo que llame este syscall otorgándole un punto.

7.4.3. Syscall 139: WhereIsMorty

Esta rutina permite calcular la distancia desde un Morty a su Rick en coordenadas relativas y es utilizada únicamente por Rick. Se decidió que el comportamiento por defecto si un Morty o un Cronenberg la usa es que no haga nada.

7.5. Debugger

El de debugger es la herramienta que se implemento para hacer un mejor seguimiento en las distintas fases de debugueo que tuvo el trabajo practico. Este permite visualizar la información de los registros justo antes de ser interrumpido por una excepción o interrupción asincrónica, como se puede ver en la Figura 8 donde se ve la pantalla del debugger justo luego de recibir una excepción del tipo `#DE` *(División por cero)*.

El funcionamiento del debugger esta controlado por el pulsado de la tecla “Y”, la cual activará el modo debug al ser presionada, y mostrará un mensaje que ha sido activado como `MODO DEBUG ON`. En caso de recibir una excepción por una tarea siendo ejecutada, se procede a saltar a la pantalla del debugger. Si la tecla “Y” es presionada nuevamente la tarea que produjo el error sera desalojada y se saltara a la tarea Idle hasta que llegue la siguiente interrupción del reloj, para así recobrar el curso de ejecución de la siguiente tarea apuntada por el scheduler. Si la tecla “Y” es presionada nuevamente desactiva este modo y en consecuencia esconde el cartelito de `MODO DEBUG ON`.

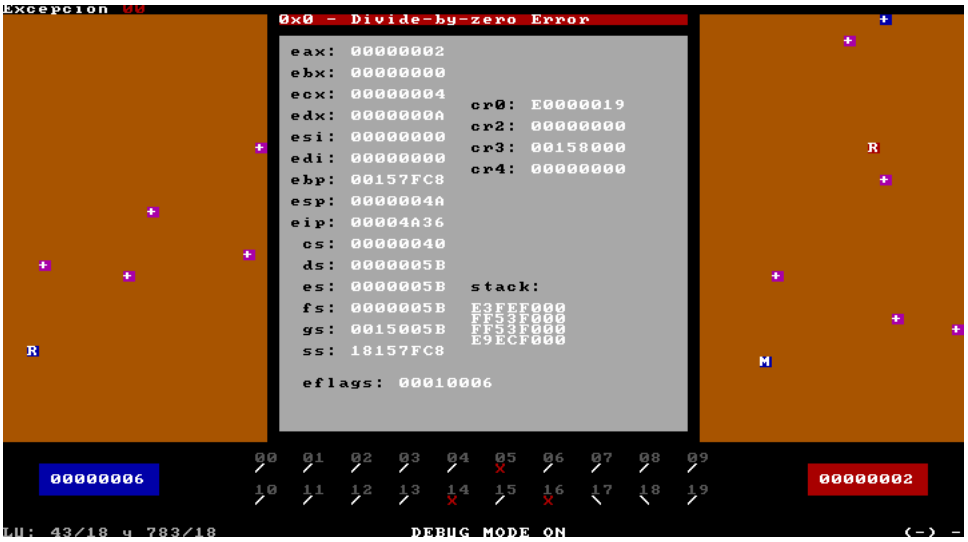

Figura 8: Captura del juego en Modo Debug

8. Conclusión

La programación de sistemas operativos o *system programming* es un área compleja fundamental en cualquier computador (y su arquitectura) que debe ser explorada de forma metódica y secuencial para evitar incontables horas de debugging. En el presente proyecto se inicio con un sistema minimal hasta construir por secciones un Kernel capaz de correr un juego con tareas concurrentes y diferentes niveles de privilegio.

El procesador de Intel provee estructuras, y estándares para la construcción de estos sistemas. En concordancia con el manual Vol.3 se segmento al Memoria en la IDT utilizando el modelo flat y luego se crearon directorios de paginas y de tablas para cada futura tarea, se mapeo memoria, se creo la IDT y sus rutinas de atención de interrupciones tanto para excepciones conocidas del procesador como interrupciones por software que permiten la dinámicas del juego o de teclado. Se crearon las TSS e inicializaron tareas con diferente nivel de privilegio al del Kernel o igual (Idle), se creo toda la lógica del backend utilizando variables globales que interactuaran con el los servicios y permitieran el correcto funcionamiento del juego, para finalmente añadir un modelo de visualización utilizando las funciones de impresión en pantalla provistas por la cátedra e ir actualizando las posiciones o estados de las tareas en cada ciclo de clock. En definitiva el código cumple todo lo pedido para ejecuta un juego con tareas simultaneas, interrupciones y rutinas de atención.