DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Grupo 15

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Leandro Rodriguez	521/17	leandro21890000@gmail.com
Miguel Rodriguez	57/19	mmiguerodriguez@gmail.com

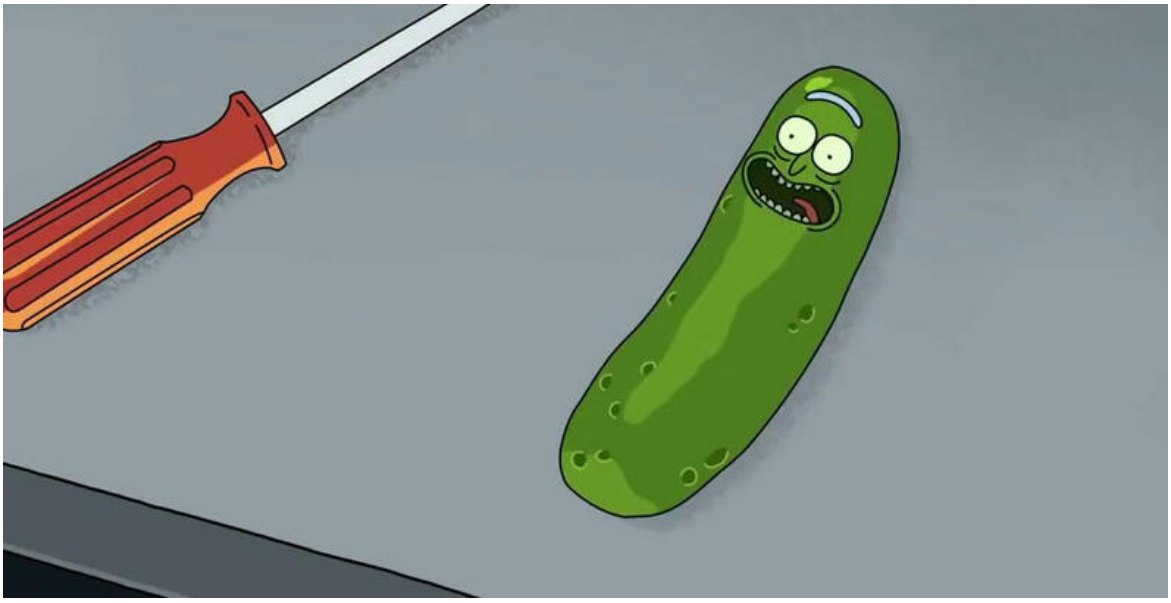
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

Este trabajo práctico consiste en la construcción de un sistema operativo básico con capacidad para ejecutar múltiples tareas en simultáneo, partiendo de una microarquitectura Intel x86 emulada por un programa llamado Bochs, y con la finalidad de implementar un juego.

En este informe se hace una explicación de todas los pasos realizados en nuestra implementación. Estos pasos van desde inicializar estructuras utilizadas por el procesador hasta funcionamiento especifico del juego.



2. Ejercicio 1: GDT, Segmentación y Modo Protegido

Al iniciar el sistema operativo, la cátedra nos proveyó con un floppy disk cuyo boot-sector ya se encontraba preparado para que el BIOS pueda realizar la carga del kernel a partir de la dirección de memoria 0x1200. De esta manera, podemos comenzar la ejecución a partir de esa dirección con el procesador en modo real.

Ahora que el kernel se encuentra cargado en memoria, nuestro primer objetivo es cambiar el modo del procesador a modo protegido, ya que para este trabajo vamos a querer realizar la construcción de un sistema operativo de 32 bits. Antes de poder pasar a podo protegido, debemos inicializar ciertas estructuras del procesador, debido a la manera en la que fue diseñado.

Una vez que se empieza a ejecutar el código contenido en `kernel.asm`, habilitamos la linea A20 para tener mayor espacio de direccionamiento (hasta 64 KB). Después, debemos cargar la GDT para poder direccionar los primeros 137 MB de memoria. Los descriptores de segmento nos permiten direccionar una posición de memoria con un selector de segmento (16 bits) y un offset (32 bits). El formato de un descriptor de segmento se puede ver en la figura 1.

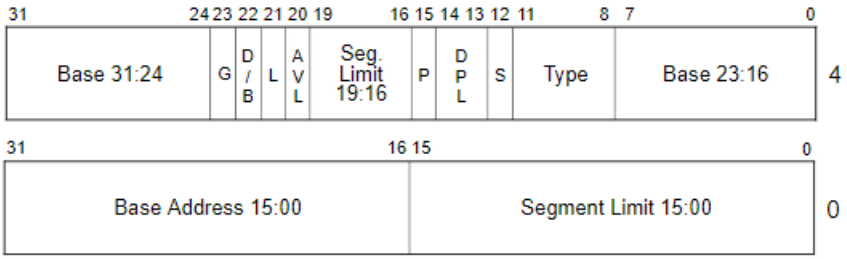


Figura 1: Descriptor de Segmento

2.1. Descriptores de segmento

Por restricciones de enunciado, debíamos inicializar los descriptores de la GDT a partir del índice 8.

En el trabajo práctico se propone la utilización de segmentación flat, ya que nos va a ser útil luego al habilitar paginación para resolver las direcciones de memoria lineales.

Definimos 2 descriptores GDT para kernel, 2 para usuario y 1 para la memoria de vídeo. Para los descriptores de kernel y usuario, debemos hacer 2 ya que uno de los descriptores será para código y el otro para datos.

Índice	Base	Límite	DPL	Tipo	G
8	0x00000000	0x088FF	0x0	0xA	1
9				0x2	
10			0x3	0xA	
11				0x2	
12	0x000B8000	0x01F3F	0x0	0x2	0

Podemos ver que los 4 descriptores tienen la misma base y límite, pero difieren en sus parámetros de tipo y DPL. El tipo define si es un segmento de código o datos, y sus permisos (lectura/escritura/ejecución). El campo DPL define sus privilegios, siendo 0 el nivel de privilegio más alto (kernel) y 3 el más bajo (usuario). Además, podemos notar que el segmento de vídeo no tiene granularidad ya que no es necesaria, mientras que los otros tienen G = 1 ya que con los 20 bits del límite no era suficiente para definir su verdadero límite. Cuando G = 1, el limite del segmento termina en (Limite * 0x1000) + 0xFFF. Luego 0x08900000 bytes = 137MB.

2.2. Salto a modo protegido

Setear el bit `PE` del registro `CR0`

```
mov eax, cr0
or eax, 0x1
mov cr0, eax
jmp 0x40:modo_protegido ; 0x40 >> 3 = 0x8. Índice en GDT.

BITS 32
modo_protegido:
```

A partir de la etiqueta `modo_protegido`, nuestro código va a estar ejecutándose en este modo.

2.3. Modo protegido

Una vez en modo protegido, resta hacer algo muy importante, que es configurar los valores de los selectores de segmentos, en particular el de datos y establecer la base de la pila de kernel. Además, le asignamos al registro `fs` el segmento de video para poder realizar la rutina para escribir la pantalla inicialmente a partir de este segmento.

```
  xor eax, eax
  mov ax, DATA0_SEGMENT
  mov ds, ax
  mov es, ax
  mov gs, ax
  mov ss, ax
  mov ax, VIDEO_SEGMENT
  mov fs, ax
```

Establecer la base de la pila

```
mov ebp, 0x27000
mov esp, 0x27000
```

2.4. Rutina para escribir/limpiar pantalla

Luego, en `map.asm` escribimos una rutina para limpiar la pantalla de video con los colores que se corresponden al color de fondo de nuestro juego.
Con motivos didácticos, el objetivo inicial de este ejercicio es hacer uso del segmento de video, para entender el funcionamiento de la unidad de segmentación al resolver direcciones lógicas a lineales.

3. Ejercicio 2 y 3: Interrupciones

Nuestro próximo objetivo en la construcción del sistema operativo, es dotar al procesador de la capacidad de poder recibir interrupciones, ya sean internas (generadas por el mismo procesador) o externas (generada por dispositivos de I/O).

En el caso de la interrupciones internas, es decir aquellas que NO provienen de algún dispositivo de entrada/salida, las atenderemos con rutinas de atención de interrupciones cada vez que ocurran para poder manejar el error asociado.

En el caso de las interrupciones 0-19, que se corresponden con las excepciones, lo que haremos es manipular la información de que llegó una interrupción, con la finalidad de transmitirla en el momento en el que queramos usar el debugger de nuestro sistema.

Para las interrupciones 137, 138 y 139 (interrupciones internas, definidas por el usuario), lo que haremos será implementar ciertas funcionalidades, que estarán relacionadas con la ejecución de las tareas y con realizar modificaciones al estado del juego, que entraremos en detalle más adelante.

Por último, para manejar las interrupciones externas correspondientes a reloj y teclado, haremos uso de un PIC, y asociaremos las interrupciones a las rutinas de atención de interrupciones 32 y 33 respectivamente. En ellas, implementaremos las funcionalidades solicitadas en el enunciado.

3.1. Entradas IDT

Para inicializar las interrupciones que van a ser atendidas utilizamos la función `IDT_ENTRY` que le indican a cada entrada de la IDT que dirección de memoria la va a atender (en nuestro caso va a ser la dirección en donde esta la función `_isrN` donde N es el numero de la interrupción. Además, el selector de segmento sera el de código de nivel kernel con atributos `DPL = 3` y `P = 1`.
Para las interrupciones 0x137, 0x138 y 0x139 los atributos serán distintos que las interrupciones por excepciones, reloj o teclado ya que son ejecutadas por código de nivel 3, por lo que su `DPL = 3`.

```
idt_entry idt[255] = { };
idt_descriptor IDT_DESC = {
    sizeof(idt) - 1,
    (uint32_t) &idt
};
```

```
#define IDT_ENTRY(numero)                                                    \
  idt[numero].offset_0_15 = (uint16_t) ((uint32_t)(&_isr ## numero) & (uint32_t) 0xFFFF);        \
  idt[numero].segsel = (uint16_t) GDT_OFF_KERNEL_CODE;                       \
  idt[numero].attr = (uint16_t) INTERRUPT_0_ATTRS;                           \
  idt[numero].offset_16_31 = (uint16_t) ((uint32_t)(&_isr ## numero) >> 16 & (uint32_t) 0xFFFF);
```

```
void idt_init() {
  IDT_ENTRY(0);
  // ...
  IDT_ENTRY(19);
  IDT_ENTRY(32); // INT Clock
  IDT_ENTRY(33); // INT Keyboard

  IDT_ENTRY(137);
  idt[137].attr = (uint16_t) INTERRUPT_3_ATTRS;
  // ...
}
```

Una vez inicializadas las estructuras para las interrupciones, resta indicarle al procesador donde esta la estructura en memoria con `lidt`, configurar el puerto del PIC para poder atender interrupciones y luego habilitar las interrupciones con `sti` que setea `EFLAGS.IF = 1`.

```
; Inicializar la IDT
call idt_init

; Cargar IDT
lidt [IDT_DESC]

; Configurar controlador de interrupciones
call pic_reset
call pic_enable

; Habilitar interrupciones
sti
```

3.2. ISR Reloj

La interrupción generada por el reloj, será atendida por la rutina de atención de interrupciones 32, que inicialmente solo imprimirá en pantalla información cada vez que ocurra un tick de reloj. Luego expandiremos el funcionamiento durante la construcción del scheduler, para realizar el cambio de tareas a partir de esta rutina.

3.3. ISR Teclado

La interrupción generada por el teclado será atendida por la rutina de atención de interrupciones 33, que inicialmente solo imprimirá en pantalla cuando el usuario presione las teclas 0-9 del teclado, pero que luego expandiremos su funcionamiento haciendo que sea un medio para acceder al mecanismo de debugging de nuestro sistema. Para mas detalle sobre esto último, ver sección 6.6.

4. Ejercicio 4 y 5: Paginación

La próxima funcionalidad que queremos agregar a nuestro sistema operativo, es habilitar el sistema de paginación del procesador. Esto nos permite, para distintas posiciones de memoria física, direccionarlas a partir de una dirección virtual (lineal). La paginación se encarga de traducir una dirección lineal a una física. Esta traducción esta ejemplificada en la figura 2.

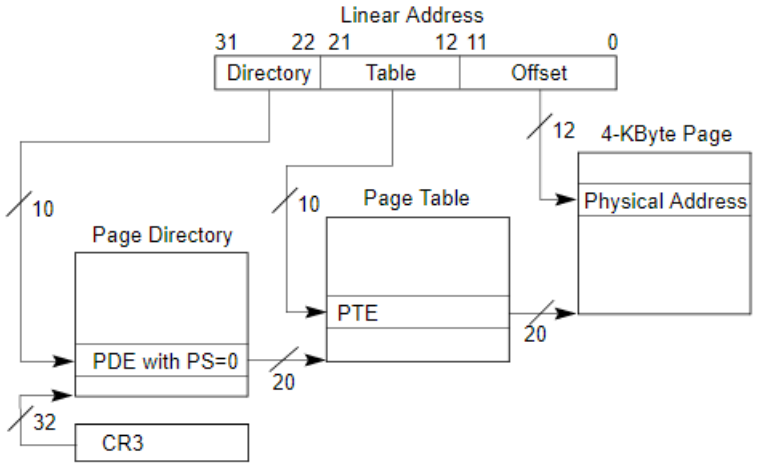


Figura 2: Traducción de dirección lineal a física con paginación

4.1. Inicialización de directorio y tablas de paginas para el kernel

El directorio de tabla de paginas del kernel debía estar situado a partir de la dirección física 0x00027000 y su primer tabla de paginas en la dirección 0x00028000. Esta tabla de paginas es utilizada para hacer un identity mapping (mapear una dirección de memoria lineal a la misma dirección física) desde la dirección 0x00000000 hasta la 0x003FFFFF (1024 * 4KB). Por lo que con una tabla podemos hacer el identity mapping de los primeros 4MB que se nos pide.

```
uint32_t mmu_initKernelDir() {
  uint32_t* page_directory = (uint32_t*) 0x00027000;
  uint32_t* page_table_0 = (uint32_t*) 0x00028000;

  for (uint32_t i = 0; i < 1024; i++) {
    page_directory[i] = 0;
    page_table_0[i] = (i << 12) | PAG_P | PAG_RW | PAG_S;
  }

  page_directory[0] = 0x00028000 | PAG_P | PAG_RW | PAG_S;

  return 0x00027000;
}
```

4.2. Activar paginación

Para habilitar la paginación, primero debemos inicializar los directorios de páginas y tablas de paginas para las direcciones lineales que vamos a querer direccionar. Una vez realizado este paso, debemos cargar en `CR3` el directorio de páginas que vamos a estar utilizando para el código que se está ejecutando actualmente (en este caso el kernel). Finalmente, resta habilitar el bit `CR0.PE = 1`. Una vez habilitado, todos los accesos a memoria van a ser resueltos con paginación.

```
; Inicializar el manejador de memoria
call mmu_init

; Inicializar el directorio de paginas
call mmu_initKernelDir

; Cargar directorio de paginas
mov cr3, eax
```

```
; Habilitar paginacion
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

4.3. Inicialización de estructuras necesarias

Para la organización de las paginas que son requeridas, lo único que usamos es una variable global `nextFreePage` que es inicializada en 0x100 desde `mmu_init` y va aumentando de a 1. Al retornar la posición de la nueva pagina, hacemos un shifteo de 12 bits ya que tenemos que aumentar de a 4KB en memoria. El sistema es muy básico, ya que siempre vamos a estar pidiendo memoria a partir de la dirección 0x100000 de a páginas de 4KB.

```
uint32_t nextFreePage;

uint32_t mmu_nextFreeKernelPage() {
  uint32_t* free_page = (uint32_t*) (nextFreePage << 12);
  for (uint32_t i = 0; i < 1024; i++) {
    free_page[i] = 0;
  }
  return ((nextFreePage++) << 12);
}

void mmu_init() {
  nextFreePage = 0x100;
}
```

4.4. Mapeo y desmapeo de memoria

Para el mapeo de una dirección virtual a una física, es necesario además recibir como parámetro el `CR3` que indica donde se va a encontrar la tabla de páginas sobre la cual vamos a insertar este mapeo. Luego, a partir de la dirección virtual, obtenemos el índice en el directorio de paginas y en la tabla de paginas. Luego, si la entrada no esta presente, entonces creamos la entrada en la tabla de paginas en una nueva pagina libre de kernel y se la seteamos al directorio de paginas del índice correspondiente.

Una vez realizado el chequeo, resta insertar en la tabla de paginas la dirección física con sus atributos correspondientes pasados por parámetro.

```
void mmu_mapPage(uint32_t cr3, uint32_t virtual, uint32_t phy, uint32_t attrs) {
  uint32_t page_directory_idx = ((virtual & 0xFFC00000) >> 22);
  uint32_t page_table_idx = ((virtual & 0x003FF000) >> 12);

  uint32_t page_directory_entry = BASE_PTR(cr3)[page_directory_idx];

  uint32_t page_table_entry = BASE_PTR(cr3)[page_directory_idx];
  if (!ENTRY_PRESENT(page_directory_entry)) {
    page_table_entry = mmu_nextFreeKernelPage() | PAG_P | PAG_RW | PAG_US;
    BASE_PTR(cr3)[page_directory_idx] = page_table_entry;
  }

  uint32_t page_entry = BASE_VALUE(phy) | attrs;
  BASE_PTR(page_table_entry)[page_table_idx] = page_entry;

  tlbflush();
}
```

4.5. Inicialización de directorio y tablas de páginas para una tarea

Para el caso de inicialización de una tarea, es necesario crear un directorio de paginas y una tabla de paginas con identity mapping. Una vez generadas las estructuras para paginación, resta identificar que tipo de tarea se va a inicializar para saber que posición física de código tenemos que copiar en el área física del Mundo Cronenberg para luego mapearla con la dirección virtual 0x08000000 y 0x08001000 (ya que los fragmentos de código de cada tarea son de 8KB).

```
uint32_t mmu_initTaskDir(enum task_type task, uint8_t x, uint8_t y) {
  uint32_t* page_directory = (uint32_t*) mmu_nextFreeKernelPage();
  uint32_t* page_table = (uint32_t*) mmu_nextFreeKernelPage();

  page_directory[0] = BASE_VALUE(page_table) | PAG_P | PAG_RW | PAG_S;
  for (uint32_t i = 0; i < 1024; i++) {
    page_table[i] = (i << 12) | PAG_P | PAG_RW | PAG_S;
  }

  // Identifico posición de la tarea
  uint32_t phy_pos1 = MAP_START + (x * TASK_SIZE) + (y * SIZE_X * TASK_SIZE);
  uint32_t phy_pos2 = phy_pos1 + PAGE_SIZE;

  // Copiar las dos paginas de la tarea
  mmu_copyPage(code1, phy_pos1);
  mmu_copyPage(code2, phy_pos2);

  // Mapear la tarea copiada
  mmu_mapPage((uint32_t) page_directory, 0x08000000, phy_pos1, PAG_P | PAG_RW | PAG_US);
  mmu_mapPage((uint32_t) page_directory, 0x08001000, phy_pos2, PAG_P | PAG_RW | PAG_US);

  return (uint32_t) page_directory;
}

void mmu_copyPage(uint32_t* code, uint32_t phy) {
  // Mapear el codigo para poder acceder a esa memoria
  mmu_mapPage(rcr3(), phy, phy, PAG_P | PAG_RW | PAG_S);

  // Copiar el codigo
  for (int i = 0; i < 1024; i++) { ((uint32_t*) phy)[i] = code[i]; }

  // Desmapear
  mmu_unmapPage(rcr3(), phy);
}
```

5. Ejercicio 6: Manejo de Tareas

Necesitamos que nuestro sistema operativo pueda generar tareas, que van a ejecutarse de manera secuencial, siempre y cuando estén vivas. Para esto, necesitamos un mecanismo que administre todo lo que ocurre cuando se deben intercambiar las tareas. Los procesadores de arquitectura x86 tienen internamente estructuras con las cuales nos proveen un mecanismo para la administración de tareas llamadas Task-State Segment (TSS). Cada tarea además, debe tener su descriptor en la tabla GDT que indica donde se encuentra en memoria física el segmento de datos de la tarea.

A diferencia de los descriptores de segmento, los descriptores para TSS usan distintos valores para algunos campos. Un ejemplo es que el bit `D/B` que antes indicaba el modo de operación, debe estar en

0 para este tipo de descriptores. Además, estos descriptores tienen un bit `BUSY` que indica si la tarea se tarea se está ejecutando actualmente.

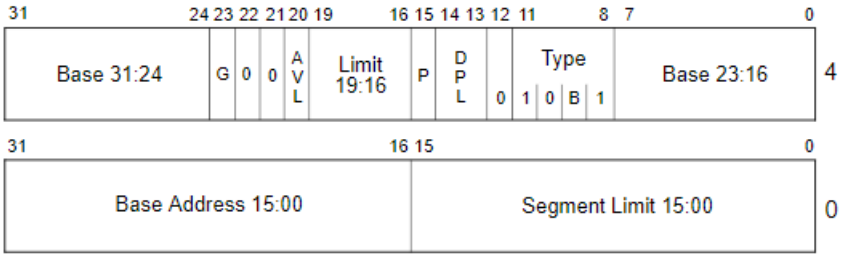


Figura 3: Descriptor de TSS

Los TSS contienen todos los datos necesarios para poder ejecutar una tarea y, en caso de haber sido desalojada la tarea, todos los datos necesarios para poder restaurar su ejecución desde donde se había detenido. En la siguiente figura se indican todos los campos guardados dentro de cada TSS.

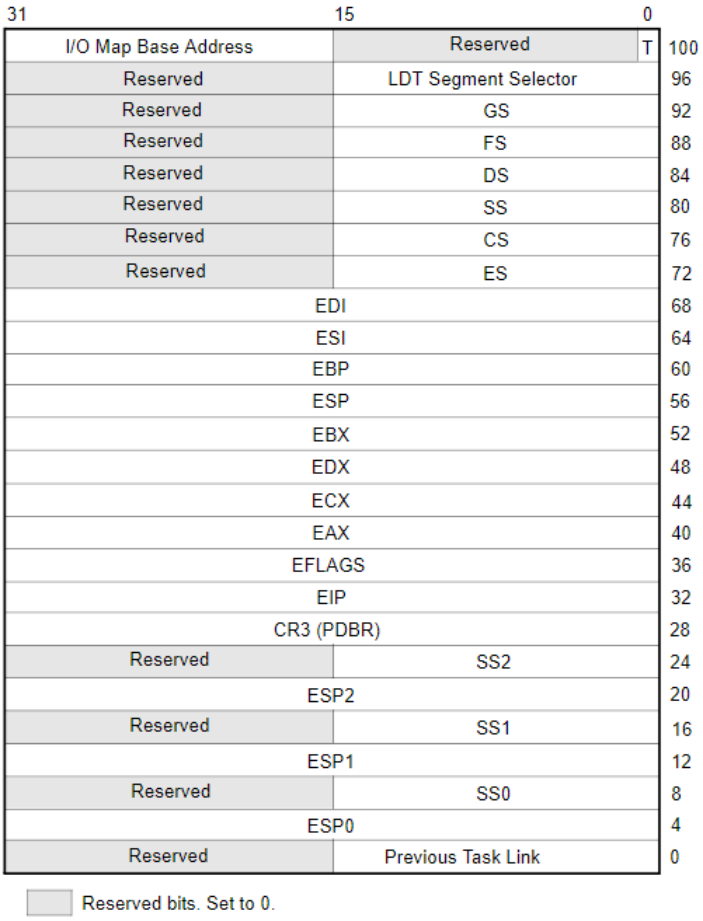


Figura 4: Task-State Segment

5.1. Descriptores de TSS

Principalmente, se necesita una tarea inicial que va a ser la primer tarea que se va a ejecutar y que va a ser instantáneamente intercambiada por nuestra tarea idle.

La inicialización de los descriptores de TSS para la tarea inicial e idle, debemos hacerla en ejecución y no estáticamente, ya que no tenemos la dirección de memoria en donde se encuentran ambos TSS.

```
tss tss_initial;
void tss_init() {
  tss_init_gdt_entry(GDT_IDX_INITIAL_TASK, (uint32_t)&tss_initial);
}
```

Esta función, la ejecutamos desde el kernel

```
call tss_init
```

5.2. Tarea idle e intercambio de tareas

Como fue explicado anteriormente, la tarea inicial únicamente nos va a servir para poder hacer el salto a la tarea idle, por lo que inmediatamente después de inicializarla, inicializaremos la tarea idle en la GDT y los datos necesarios en su TSS.

```
void tss_init_idle_task() {
  tss_init_gdt_entry(GDT_IDX_IDLE_TASK, (uint32_t)&tss_idle);

  tss_idle.esp    = 0x00027000;
  tss_idle.ebp    = 0x00000000;
  tss_idle.eip    = 0x0001A000;
  tss_idle.cr3    = 0x00027000;
  tss_idle.es     = GDT_OFF_KERNEL_DATA;
  tss_idle.cs     = GDT_OFF_KERNEL_CODE;
  tss_idle.ss     = GDT_OFF_KERNEL_DATA;
  tss_idle.ds     = GDT_OFF_KERNEL_DATA;
  tss_idle.fs     = GDT_OFF_KERNEL_DATA;
  tss_idle.gs     = GDT_OFF_KERNEL_DATA;
  tss_idle.eflags = 0x00000202;
  tss_idle.iomap  = 0xFFFF;
```

Por el enunciado, se nos indica que la tarea idle, se encuentra en la dirección de memoria física 0x0001A000 (`EIP`), que su pila se alojara en la misma dirección que la pila del kernel (`ESP`) y que además, debe compartir el mismo `CR3` que el kernel. Al ser código que va a ser ejecutado por el kernel, los segmentos de datos y de código serán los de nivel 0. Para el campo `EFLAGS` activamos el bit `IF` (Interrupt Flag) para habilitar interrupciones desde esta tarea.

Al igual que con la tarea inicial, este código es ejecutado desde el kernel.

```
call tss_init_idle_task
```

Posteriormente, debemos cargar la tarea inicial. Para esto utilizamos la instrucción `ltr`.

```
mov ax, TSS_INIT_SEGMENT
ltr ax
```

Finalmente, para realizar el intercambio entre la tarea inicial y la idle, tenemos que hacer un `jmp far` hacia la primer instrucción de la tarea idle. Para esto, utilizamos su selector de segmento que fue previamente inicializado en la GDT.

```
jmp TSS_IDLE:0x0
```

5.3. Rutina para completar una TSS

Al inicializar tareas, estas tienen un tipo (Rick137, RickD248, MortyC137, MortyD248 o Cronenberg) y una posición x e y sobre la cual van a estar posicionadas en el Mundo Cronenberg. Además, internamente, nosotros nos guardamos su índice en el que lo insertamos en la tabla GDT.

Cuando nosotros queramos crear una tarea, debemos pedir memoria en donde vamos a guardar su TSS, su pila de nivel 0 e inicializar su directorio de paginas, junto con la copia de código con `mmu_initTaskDir`. Además, debemos crear una entrada en la tabla GDT para cada nueva tarea que creemos.

```
uint32_t tss_init_task(enum task_type task, uint32_t index_gdt, uint8_t x, uint8_t y) {
  tss* task_tss = (tss*) mmu_nextFreeKernelPage();
  uint32_t kernel_stack =  mmu_nextFreeKernelPage();
  uint32_t task_page_directory = mmu_initTaskDir(task, x, y);

  tss_init_gdt_entry(index_gdt, (uint32_t) task_tss);

  (*task_tss).esp0    = kernel_stack + 0x1000;
  (*task_tss).ss0     = GDT_OFF_KERNEL_DATA;

  (*task_tss).cr3     = task_page_directory;
  (*task_tss).eip     = 0x08000000;
  (*task_tss).eflags  = 0x00000202;
  (*task_tss).iomap   = 0xFFFF;

  (*task_tss).es      = GDT_OFF_USER_DATA | 0x3;
  (*task_tss).cs      = GDT_OFF_USER_CODE | 0x3;
  (*task_tss).ss      = GDT_OFF_USER_DATA | 0x3;
  (*task_tss).ds      = GDT_OFF_USER_DATA | 0x3;
  (*task_tss).fs      = GDT_OFF_USER_DATA | 0x3;
  (*task_tss).gs      = GDT_OFF_USER_DATA | 0x3;

  return task_page_directory;
}
```

Notemos que esta función retorna el directorio de paginas de la tarea que inicializamos. Esto será utilizado para guardar el `CR3` de una tarea ya que es necesario para algunas funcionalidades del sistema.

6. Ejercicio 7: Scheduler

Por último, queremos proveer al sistema operativo la funcionalidad de poder ejecutar múltiples tareas al mismo tiempo (multitasking), que es el término con el que es conocida la funcionalidad, aunque realmente en la implementación se trata de poder realizar ejecuciones secuenciales.

El diseño estructural que designamos para nuestro scheduler será de tipo round-robin. Es decir, asignaremos proporciones de tiempo iguales para la ejecución de cada una de las tareas , y de esta manera lograríamos que nuestro sistema soporte multitasking.

La cantidad de tiempo que asignaremos a cada tarea es la correspondiente a 1 tick de clock, uno de los dispositivos de E/S que interrumpirá a nuestro sistema, y cuya rutina de atención de interrupciones para atender dicha interrupción será la 32, como ya vimos en la sección 3.2.

6.1. Inicialización

Para la inicialización de nuestro scheduler, lo primero que haremos será llamar a una función que se encargue de inicializar las estructuras de datos que utilizaremos para manejar su estado.

```
; Inicializar el scheduler
   call sched_init
```

En principio, las estructuras que utilizaremos en el scheduler serán: un índice de tarea, para tener el índice correspondiente al descriptor de TSS de la tarea asociada que se encuentra cargada actualmente, y una máscara para saber si cada una de las tareas se encuentra activa en el sistema o debe ser desalojada (entendiendo a una tarea desalojada como aquella que ya no es alcanzada por el scheduler) y por lo tanto nunca más ejecutará su contenido.

```
 void sched_init() {
  current_task_idx = idle_task_idx;
  for (uint8_t i = 0; i < 25; i++) {
    task_alive[i] = TRUE;
  }
}
```

6.2. Siguiente tarea

Como mencionamos anteriormente, dentro de la rutina de atención de interrupciones de reloj es donde realizaremos el task switch, y para hacerlo necesitamos una función que designe cual será la próxima tarea a ser ejecutada. Observar que hacemos uso de las estructuras que habíamos designado para el scheduler, y que tenemos ciertos booleanos en nuestro sistema para saber si el juego terminó o si se encuentra en modo debug, tal que si ocurriese cualquiera de ambos casos, la próxima tarea a ser ejecutada sería la idle, como fue designado en el enunciado.

Como observación, vemos que solo podemos pasar a tareas que se encuentren vivas en el sistema, es decir que no hayan sido desalojadas previamente.
Es dentro de la siguiente implementación, donde efectivamente hacemos que el scheduler actúe con el comportamiento de un round-robin, ya que simplemente lo que hacemos es pasar a la próxima tarea.

```
int16_t sched_nextTask() {
  if (debug_mode == FALSE && game_finished == FALSE) {
    current_task_idx++;
    if (current_task_idx > max_task_idx) {
      current_task_idx = init_task_idx;
    }
    if (task_alive[SCHED_TASK_IDX(current_task_idx)] == FALSE) {
      return sched_nextTask();
    }
    return (current_task_idx << 3);
  }

  return (idle_task_idx << 3);
}
```

6.3. Modificación int 0x137, 0x138 y 0x139

La implementación de las rutinas de atención de interrupciones 137, 138 y 139 será útil para habilitar funcionalidades dentro de nuestro juego. Se tratan de servicios que proveerá el sistema operativo para poder cambiar el estado del juego. Nuestra implementación de estos servicios, no tienen contemplada la preservación de todos los registros, ya que en el enunciado no estaba especificado que comportamiento dar al realizar las llamadas.

Son funcionalidades particulares y tienen que ver con el funcionamiento del juego en sí, motivo por el cual creamos funciones que realicen esos cambios en `C`, y las llamamos desde las rutinas como se ve a continuación:

```
_isr137:
  push edx
  push ecx
  push ebx
  push eax
  call game_usePortalGun
```

```
  add esp, 4 * 4
  iret

_isr138:
  push ax
  call game_IamRick
  add esp, 2
  iret
```

A diferencia de las otras subrutinas, que requerían retornar el resultado en un sólo registro, en la syscall de whereIsMorty para poder implementar el funcionamiento para que se retorne en `eax` y `ebx` lo que hacemos es pasar por parámetro un puntero a la pila, que va a ser modificada y luego hacemos un `pop` de la memoria en la pila hacia los registros.

```
_isr139:
  sub esp, 4 * 2
  push esp
  call game_whereIsMorty
  add esp, 4 * 1
  pop eax
  pop ebx
  iret
```

Para mas detalle sobre la implementación del estado del juego y su funcionamiento, ver sección 7.

6.4. Intercambio de tareas

Como habíamos designado, en cada tick de reloj debemos transicionar de la tarea actual a la tarea siguiente y proceder a la realización del task switch. Observar también, que aquí es donde decidimos llamar a funciones auxiliares que realicen acciones tales como cambios en el estado del juego, o dibujado por pantalla para poder ver el funcionamiento del mismo.

```
;; Rutina de atencion del RELOJ
_isr32:
  pushad
  call pic_finish1

  ; Si el juego termino, retornar
  cmp byte [game_finished], 1
  je .fin

  call game_enablePortalGuns
  call nextClock
  call screen_incTaskClocks
  call screen_drawTasks
  call game_checkEndOfGame

  ; Proxima tarea a ejecutar
  call sched_nextTask
  str bx
  cmp ax, bx
  je .fin

  mov word [sched_task_selector], ax
  jmp far [sched_task_offset]
.fin:
```

```
  popad
  iret
```

6.5. Desalojo de tarea en excepción

Cuando cuando una tarea produzca una excepción, la misma sea desalojada del sistema, es decir, que no pueda volver a ejecutarse.

Para realizar esto, debemos modificar la estructura de nuestro scheduler, en particular el array de máscaras que nos permitía saber si una tarea se encuentra viva o muerta. De esta manera, como ya habíamos comentado en la sección 6.1, y visto en la sección 6.2 la tarea quedará desafectada del sistema.

6.6. Debugging

Nuestro sistema contará con un mecanismo de debugging, que utilizará una interrupción de teclado para ser invocado. Mas específicamente, si durante la ejecución de nuestro juego presionamos la tecla ”Y”, se activará o desactivará el modo de debugging el cual podemos ver representado en la figura 5.

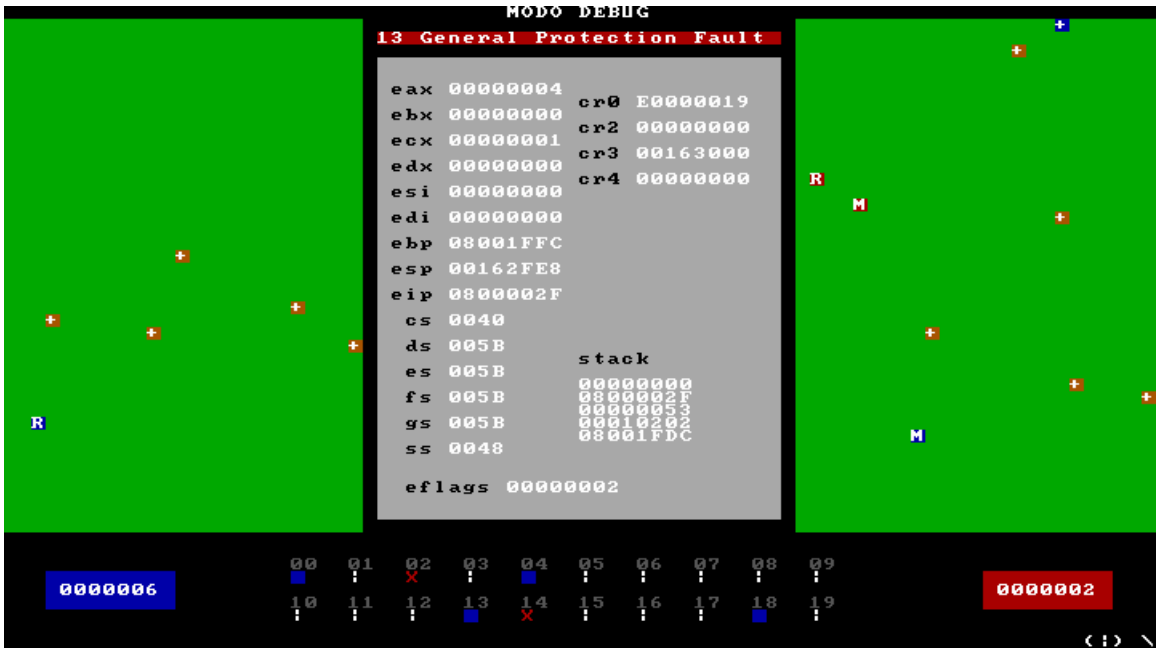


Figura 5: Modo Debug

En la figura podemos apreciar el estado de los registros del procesador al ocurrir una excepción.

Para realizar esto, simplemente en la rutina de atención de interrupciones presentada en el la sección 3, tenemos una macro para manejar las excepciones de manera genérica:

```
_isr%1:
  cmp byte [debug_mode], 0 ; Si el modo debug no esta activo
  je .skip_excep%1 ; No mostramos el modo debug
  pushad
  mov eax, [esp + 36] ; EIP
  push eax
  pushfd ; EFLAGS
  push cs
  push ds
  push es
```

```
  push gs
  push ss
  push fs
  mov eax, cr0
  push eax
  mov eax, cr2
  push eax
  mov eax, cr3
  push eax
  mov eax, cr4
  push eax
  mov eax, %1 ; Tipo de excepcion
  push eax
  push esp
  call game_save_exception_info
  call game_show_debug_info
  .skip_excep%1:
  call game_kill_current_task
  call sched_kill_current_task ; Mata la tarea actual, salta a la idle
```

Es aquí, cuando el modo debug se encuentra activado, donde presentamos la información de la excepción en nuestro el juego, caso contrario, solo dejamos desafectada la tarea.

7. Ejercicio 8: Servicios del sistema

En esta sección expandiremos un poco más en detalle como implementamos las syscalls que provee nuestro sistema para tener flexibilidad a la hora de crear las tareas para nuestro juego.

Para el desarrollo de nuestro juego, tenemos ciertas estructuras que utilizamos para manejar toda la información que necesitemos. La idea es que en cada uno de los syscalls, hacemos uso de las estructuras y proveemos la funcionalidad solicitada.

```
struct pos {
  uint8_t x;
  uint8_t y;
};

struct task {
  struct pos portal;
  struct pos position;
  e_taskType_t type;
  uint32_t cr3;
  char alive;
  char used_portal;
  uint8_t count_portal;
};

struct game {
  struct task task[25];
  char has_cronenberg[SIZE_X][SIZE_Y];
  int16_t cronenberg_idx[SIZE_X][SIZE_Y];
  uint32_t minds_rick_c137;
  uint32_t minds_rick_d248;
};
```

Las estructuras son inicializadas desde el kernel de la siguiente manera:

```
; Inicializar game
call game_init
```

A efectos de la realización del informe, no queremos entrar muy en detalle en la implementación de los llamados al sistema, sino que solo presentaremos las decisiones más importantes que tomamos para el realizamiento de las funcionalidades solicitadas.

Pensar también que, la implementación comprende también acciones como realizar llamados a print/unprint de la pantalla, modificar el contenido de las estructuras del scheduler, manejar el debugging y lógica correspondiente para sumar puntos, chequear si finaliza o no el juego, que si bien son acciones significantes, su desarrollo no tiene que ver con decisiones arquitecturales a nivel de nuestro sistema, que es lo que nos interesa presentar en el informe.

7.1. Syscall usePortalGun

La decisión mas importante durante el desarrollo de esta sección, fue cuando necesitamos mover la posición del Morty junto a la del Rick, en el caso en el que el Rick use el arma de portales con el valor de withMorty en 1, es decir un llamado a `usePortalGun(x, y, z, 1)`.

En este caso, ocurre que el contexto de ejecución del procesador se encuentra con el `CR3` correspondiente a la tarea del Rick que llama a la función, pero necesitamos acceder al `CR3` del Morty asociado a ese Rick, para poder realizar el mapeo correspondiente a su código en el destino indicado.

Para solucionar esto, lo que hicimos fue guardar en la estructura del juego los `CR3` de cada tarea, así de esta manera podemos realizar el mapeo llamando a la función implementada en la sección 4.4:

```
void mmu_mapPage(uint32_t cr3, uint32_t virtual, uint32_t phy, uint32_t attrs);
```

Observar que pudimos hacer esto, porque en el momento en el que estamos accediendo a las syscalls, tenemos privilegios del nivel de sistema operativo, ya que fuimos llamados luego de haberse producido una interrupción de reloj. Esto nos permite no tener ningún error de permisos, ya que no somos la tarea que está ejecutando, somos el sistema operativo.

Una vez solucionado ese tema, el resto de la implementación es bastante directa, se trata de modificar el estado del juego según indique el enunciado, utilizando los parámetros de la función como corresponde.

7.2. Syscall IamRick

En este caso del desarrollo fue bastante directo también, se trató de implementar las funcionalidades solicitadas, sin ningún cambio notorio o significante en las estructuras pensadas para el juego.

7.3. Syscall whereIsMorty

De igual manera que en la syscall anterior, no tuvimos que realizar ningún cambio significativo durante el desarrollo de esta llamada al sistema.

8. Conclusiones

En el presente trabajo se analizaron las herramientas y disponibilidades propuestas por un procesador Intel x86 para la construcción de un sistema operativo de 32 bits.

El trabajo fue realizado siguiendo una construcción metódica, donde poco a poco fuimos aprendiendo conceptos a medida que íbamos probando efectivamente que funcionasen en nuestro sistema las implementaciones propuestas, junto al desafío de ir agregando funcionalidades nuevas para terminar con el objeto resultante: un sistema básico, funcional y flexible.