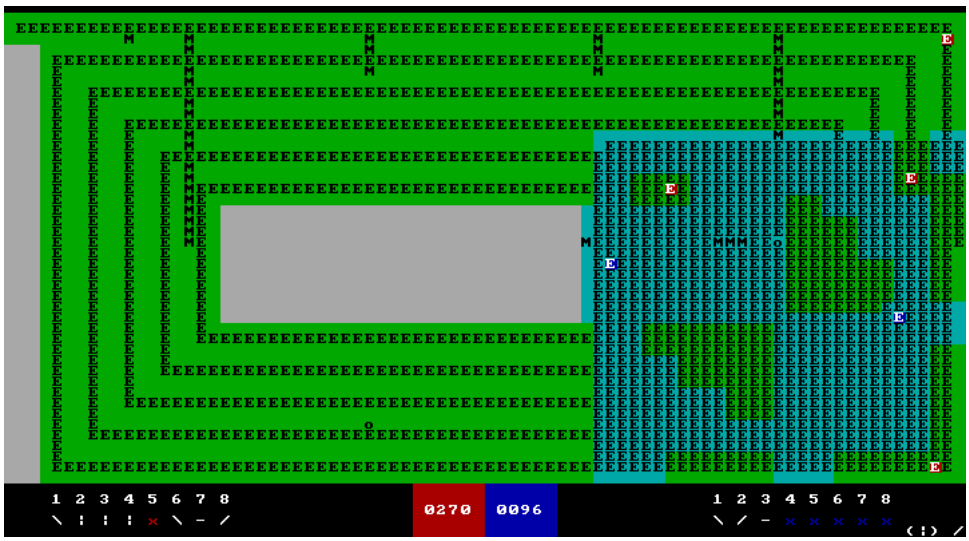
Organización del Computador II

TP3

Tierra Pirata

20 de junio de 2015



Integrante	LU	Correo electrónico
Christian Cuneo	755/13	chriscuneo93@gmail.com
Julián Bayardo	850/13	julian@bayardo.com.ar
Martin Baigorria	575/14	martinbaigorria@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El objetivo del presente trabajo practico es aprender y aplicar diferentes conceptos de *System Programming*. A partir de una implementación de un boot-sector, se programo un pequeño kernel con los diferentes mecanismos de protección y ejecución concurrente de tareas para luego poder ejecutar un juego con hasta 16 tareas concurrentes a nivel de usuario.

1.1. Inicialización

Al prender la computadora, comienza la inicialización del `POST` (Power-On Self-Test), un programa de diagnostico de hardware que verifica que todos los dispositivos se han inicializado de manera correcta. Una vez terminado el `POST`, el `BIOS` se encarga de identificar el primer dispositivo de booteo, ya sea un CD, un disco rígido o un diskette. En este trabajo, inicializaremos el sistema a partir de un diskette.

El `BIOS` (Basic Input-Output System) copia de memoria `RAM` los primeros 512 bytes del sector a partir de la dirección `0x7c00` de un diskette. Esto se copia comenzando en la dirección `0x1200` y luego se ejecuta el boot-sector a partir de allí. El boot-sector encuentra en el floppy el archivo `kernel.bin` , y luego lo copia en memoria a partir de la dirección `0x1200` , ejecutando a partir de la misma.

2. Modo Real

2.1. Introducción

Por una cuestión de compatibilidad hacia atrás, al inicializar un procesador Intel, el mismo funciona como un 8086, lo que conocemos como `Modo Real`. En este modo, no existe la protección por hardware, es decir, cualquier código en ejecución tiene acceso a todos los segmentos de memoria y puede utilizar cualquier instrucción del 8086. Para poder utilizar otras instrucciones y funcionalidades mas avanzadas, también habilitando la protección por hardware, se debe pasar a Modo Protegido.

2.2. A20

El addressing line `A20` forma parte del bus de direcciones del procesador. En un `8086`, este bus tiene 20 lineas, numeradas de la 0 a la 19. Sin embargo, cuando salio al mercado el `80286`, el primero en soportar el modo protegido, el bus de direcciones paso a tener 24 bits. El problema que surgió es que muchos programadores en su código del `8086` utilizaban lo que se conoce como wrap-around. Es decir, cuando accedían a memoria, utilizaban el overflow en el bus de direcciones como parte de la lógica de sus programas. El 80286 no soportaba este overflow, rompiendo la compatibilidad hacia atrás, dado que tenia 4 lineas de address adicionales.

Para solucionar este problema, a IBM se le ocurrió utilizar un pin del controlador del teclado que estaba sin usar y conectarlo a la linea 20 del bus de direcciones para poder forzar el overflow en los programas viejos. Por esta razón, antes de pasar a modo protegido se debe habilitar esta linea, para poder utilizar todo el espacio direccionable por todas las lineas del bus de direcciones. Esto se realiza a través de un handshake que consiste en enviar una sequencia de bytes particular por la linea de teclado.

2.3. Global Descriptor Table

Antes de poder pasar a modo protegido, debemos configurar de GDT. La GDT es un arreglo en memoria de hasta 8192 entradas de 8 bytes cada una, cada una de estas entradas define la configuración de un segmento de memoria, o de una tarea. En nuestro código, puede encontrarse en `gdt.h` la estructura de `gdt_entry` con sus respectivos parámetros de configuración. La figura de aquí abajo muestra el significado de los campos para un descriptor de segmento.

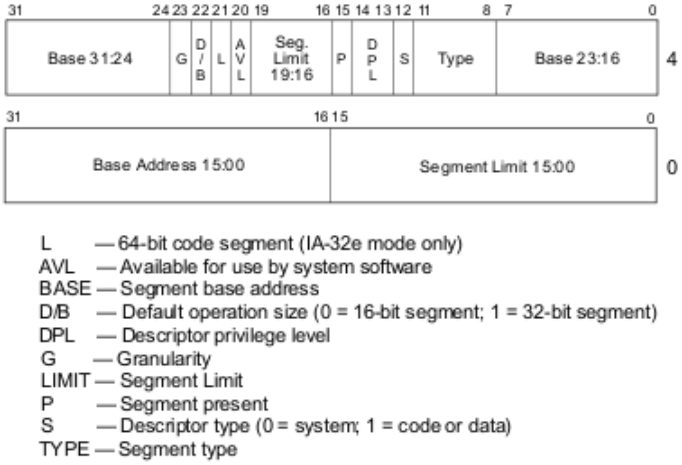


Figura 1: Segment Descriptor

En su primera encarnación, la GDT nos permite configurar permisos de acceso a la memoria, utilizando el conocido modelo de memoria segmentado. En nuestro caso particular, nosotros hacemos un identity mapping de los primeros 500 MB de memoria (que por otro lado es más de lo que utilizamos), evitando de esta forma tener que realmente tener en cuenta los efectos de segmentación. En concreto, el hacer un identity mapping nos permite "deshabilitar"la segmentación: a pesar de que no sea posible a nivel procesador desactivarla, y el proceso de traslación de direcciones se realiza igual, no tenemos efectos colaterales. Más adelante se extendió la capacidad de la GDT para soportar el entorno multitarea que conocemos hoy en día, pero trataremos sobre ese tema más adelante.

Para nuestro kernel habilitamos un total de 5 segmentos de memoria fuera del NULL segment y los 8 reservados por la cátedra. Los 5 segmentos corresponden a, respectivamente: código y datos con nivel de protección 0, código y datos con nivel de protección 3, y video con protección 0 (observemos que esto sólo lo utilizaremos más adelante, cuando activemos modo protegido). Como dijimos antes, los 4 segmentos que no corresponden a vídeo están mapeados sobre el mismo espacio de memoria, cambiando únicamente el nivel de privilegio requerido para acceder. Esta elección permite que el esquema de memoria no cambie a lo largo del código, ayudando a la comprensión y correctitud del código. En tanto al segmento para video, este mapea las direcciones específicas del frame buffer, y le damos acceso sólo a nivel 0, pudiendo así evitar que las tareas del usuario cambien los datos de video.

Luego de haber configurado la GDT, la instrucción `lgdt` nos permite cargar un descriptor de GDT indicando la ubicación de la GDT en la memoria para que el procesador pueda comenzar a aplicar segmentación como corresponde. Este descriptor contiene la dirección física de la GDT, así como el tamaño de la estructura. Cabe destacar que el espacio reservado para la GDT es de 32 entradas: 5 correspondientes a las del kernel, 1 del NULL descriptor, 8 reservadas por la cátedra, y el resto destinadas al manejo de tareas.

2.4. Pasaje a Modo Protegido

Una vez armada la GDT y habilitado la puerta A20, debemos habilitar Modo Protegido. El modo de protección esta definido por el bit menos significativo del registro `CR0` . Usando un OR con `0x1` , habilitamos este bit.

Una vez que tenemos todas las estructuras necesarias armadas, hay que hacer un `jmp far` al segmento de código de nivel 0 en la `GDT`. De esta forma finalmente habilitamos la protección por hardware y pasamos a Modo Protegido. A partir de aquí es fundamental poner los selectores de segmentos correspondientes a los datos y video, habilitándonos efectivamente para el uso de memoria y el frame buffer. Cabe destacar, además, que no necesariamente tenemos una pila que podamos utilizar, por lo que es importante poner los registros correspondientes en valores que nos sean útiles.

3. Modo Protegido

3.1. Niveles de protección

Intel soporta 4 niveles de protección diferentes, siendo 0 el mas alto y 3 el mas bajo. Por esa razón los bits de protección tienen 2 bits.

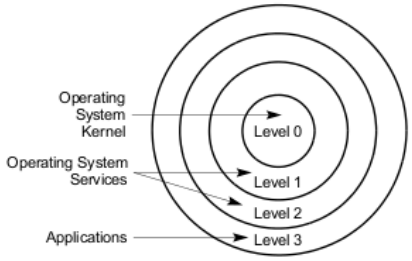


Figura 2: Protection Rings

Si el $max(CPL, RPL) > DPL$ (recordemos que un nivel de protección mayor numérico se corresponde a un menor nivel de privilegio) al querer acceder o hacer un salto a un segmento, la `Unidad de Protección` verifica que no tenemos privilegios suficientes y el procesador nos da un General Protection Fault (#GP). Esto se puede ver en la siguiente figura:

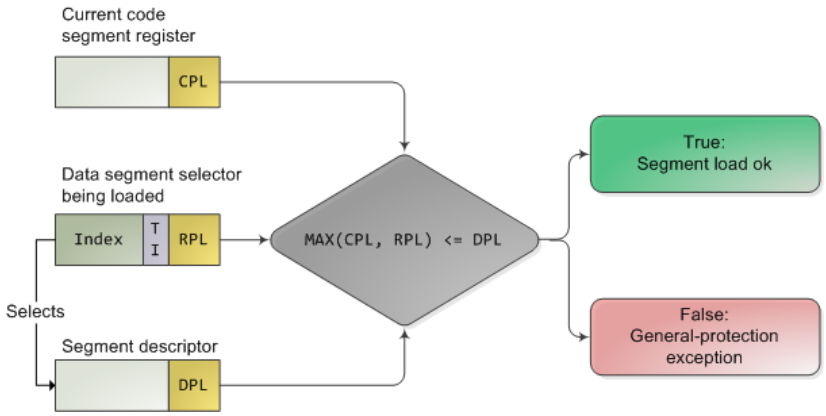


Figura 3: Protection Diagram

3.2. Memory Management Unit

Un procesador Intel, para gestionar lo que son los accesos a memoria, utiliza una `MMU` (Memory Management Unit). La misma esta compuesta por la Unidad de Segmentación y la Unidad de Paginación. La siguiente figura ilustra la idea general:

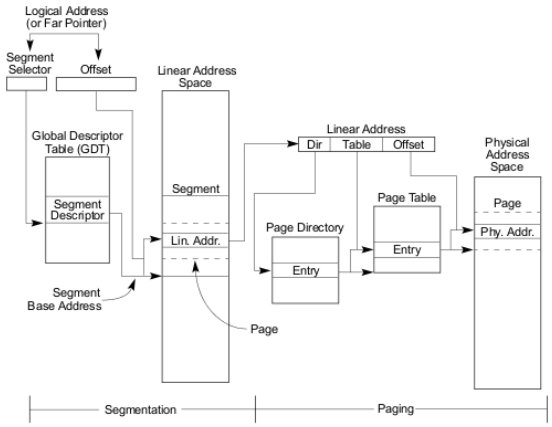


Figura 4: Segmentation & Paging

3.2.1. Unidad de Segmentación

La unidad de segmentación se ocupa de pasar desde las *direcciones lógicas* a *direcciones lineales*. Su comportamiento está determinado por los *registros de segmentación* (todos los que terminan en s), los mismos deben cargarse con el offset en la GDT del descriptor de segmento, permitiéndole a la CPU identificar el segmento adecuado dado un address lineal, la unidad de protección se ocupará de verificar que el `DPL` es compatible con el `CPL` y el `RPL`.

Por lo tanto, al entrar a modo protegido, lo primero que hacemos es inicializar los diferentes selectores de segmento. Los mismos tienen el siguiente formato:

Figura 5: Segment Selector

El index corresponde al descriptor de segmento en la `GDT`. Nosotros no utilizaremos la LDT, por lo que el bit TI estará siempre en 0. Ademas el `RPL` (Requested Protection Level) estará siempre en nivel 0 (superuser) o en 3 (usuario).

En modo protegido, los selectores de segmento tienen 16 bits. Los 13 bits mas significativos contienen el índice dentro de la tabla de descriptores. El bit 2 especifica si la operación utiliza la `GDT` o la `LDT`. Finalmente, los 2 bits menos significativos definen el nivel de privilegio solicitado.

3.2.2. Unidad de Paginación

Para activar la paginación, en primer lugar debemos inicializar el directorio de paginas y cargar el registro `cr3` con la dirección del mismo. Como los directorios de paginas están alineados a 4 kB, los primeros 12 bits del `cr3` no son necesarios para identificar el directorio, por lo que son utilizados por atributos del procesador. En nuestro caso no utilizamos estos atributos, por lo que son todos 0. La siguiente tabla muestra el formato del `cr3` .

Bit Position(s)	Contents
2:0	Ignored
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page directory during linear-address translation (see Section 4.9)
11:5	Ignored
31:12	Physical address of the 4-KByte aligned page directory used for linear-address translation
63:32	Ignored (these bits exist only on processors supporting the Intel-64 architecture)

Figura 6: CR3 Format

Luego activamos la paginación con el ultimo bit del registro `cr0` . Una vez activada, la dirección lineal comienza a pasar luego por la unidad de paginación. La unidad de paginación se encarga de ir desde la dirección lineal a la dirección física en memoria. En caso de que la dirección lineal no este paginada, el procesador tiene una Page Fault Exception (#PF).

Para facilitar el manejo y armado de estructuras de paginación, creamos varias funciones en C. Como sabemos, cada directorio de paginas tiene 1024 entradas de descriptores de 4 bytes. Lo mismo sucede con los directorios de paginas, que también tienen 1024 entradas con descriptores de 4 bytes.

Bit Position(s)	Contents
0 (P)	Present; must be 1 to reference a page table
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-MByte region controlled by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-MByte region controlled by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the page table referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether this entry has been used for linear-address translation (see Section 4.8)
6	Ignored
7 (PS)	If CR4.PSE = 1, must be 0 (otherwise, this entry maps a 4-MByte page; see Table 4-4); otherwise, ignored
11:8	Ignored
31:12	Physical address of 4-KByte aligned page table referenced by this entry

Figura 7: Page Directory Entry

Bit Position(s)	Contents
0 (P)	Present; must be 1 to map a 4-KByte page
1 (R/W)	Read/write; if 0, writes may not be allowed to the 4-KByte page referenced by this entry (see Section 4.6)
2 (U/S)	User/supervisor; if 0, user-mode accesses are not allowed to the 4-KByte page referenced by this entry (see Section 4.6)
3 (PWT)	Page-level write-through; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
4 (PCD)	Page-level cache disable; indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9)
5 (A)	Accessed; indicates whether software has accessed the 4-KByte page referenced by this entry (see Section 4.8)
6 (D)	Dirty; indicates whether software has written to the 4-KByte page referenced by this entry (see Section 4.8)
7 (PAT)	If the PAT is supported, indirectly determines the memory type used to access the 4-KByte page referenced by this entry (see Section 4.9.2); otherwise, reserved (must be 0)[1]
8 (G)	Global; if CR4.PGE = 1, determines whether the translation is global (see Section 4.10); ignored otherwise
11:9	Ignored
31:12	Physical address of the 4-KByte page referenced by this entry

Figura 8: Page Table Entry

El procesador, al buscar estas estructuras en memoria RAM, requiere que las mismas estén alineadas a 4 kB, dado que es el tamaño de pagina que carga en memoria cache. A continuación, una breve explicación de qué hacen las funciones exportadas al usuario (cabe destacar, hay numerosos comentarios en el código explicando las particularidades):

- `int mmap(uint virtualAddress, uint physicalAddress, uint directoryBase, uchar readWrite, uchar supervisorUser)` : mapea la página donde se encuentra el address virtual a la página donde se encuentra el address físico, indicando permisos de escritura o de usuario según se indique por parámetro.
- `int munmap(uint directoryBase, uint virtualAddress)` : desmapea la página donde se encuentra el address virtual.
- `int remap(uint directoryBase, uint virtualAddress, uint physicalAddress)` : Remapea la página donde se encuentra el address virtual a la página donde se encuentra el address físico.
- `int isMapped(uint directoryBase, uint virtualAddress)` : Devuelve true si, y solo si, la página donde se encuentra el address virtual está presente, es decir, si el address virtual es accesible.
- `int mmu_move_codepage(uint directoryBase, uint codeBaseSrc, uint codeBaseDst)` : Copia 1 página de memoria (4 KB) desde el primer address virtual hacia le segundo address virtual. Es importante tener en cuenta los page boundaries, porque el código va a generar un page fault si no hay suficiente espacio.

No es necesario inicializar ninguna estructura particular fuera de estar a nivel kernel para utilizar estas funciones. Es importante marcar que por el esquema de memoria que armamos, estas funciones pueden actuar sobre cualquier directorio de tablas de página, aun sin ser el propio. Es decir, no es necesario cambiar el `cr3` para manipular la paginación de una tarea, sino que basta con tener acceso de escritura al directorio y tablas de páginas correspondientes.

3.3. Interrupt Descriptor Table

Una interrupción es una señal que le indica a la CPU que debe interrumpir la ejecución actual de instrucciones. El rol de la `IDT` (Interrupt Descriptor Table) es contener los diferentes descriptores de interrupción y asociar las diferentes interrupciones a sus respectivas rutinas de atención de interrupción. Existen tres fuentes de interrupciones:

1. Hardware: donde la interrupción proviene de un dispositivo conectado directamente al procesador, como por ejemplo el timer, o el teclado.

2. Software: en las que el mismo software llama al sistema, esto se utiliza para poder hacer cambios de nivel de privilegio, permitiendo al kernel exponer un conjunto de servicios común a todos los programas corriendo en la máquina.

3. Internas: que pertenecen estrictamente al procesador y ocurren, por ejemplo, cuando hay errores graves. La excepción de división por 0 es uno de estos casos.

A su vez, la `IDT` puede contener tres tipos de descriptores:

1. Interrupt Gate

2. Trap Gate

3. Task Gate

Para construir la IDT, armamos primero en C la estructura `idt_entry` con sus respectivos atributos y luego construimos un array de 256 posiciones del mismo (la máxima cantidad soportada por el PIC). En este trabajo practico solo utilizaremos descriptores de interrupción y de tarea.

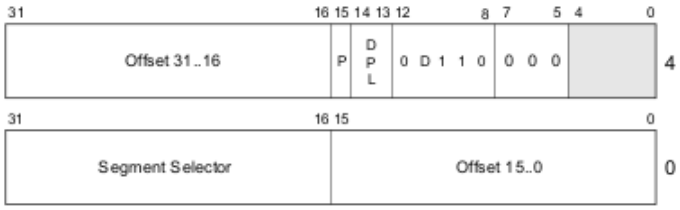


Figura 9: IDT Descriptor

Modificamos la macro de la cátedra para poder cargar la IDT con diferentes atributos. Luego inicializamos las diferentes posiciones que utilizamos con sus respectivos selectores de segmento y atributos, tomando también la referencia a las respectivas rutinas de atención.

Hay que tener mucho cuidado al settear los atributos. Caso contrario, al cambiar de segmento podemos tener un General Protection Fault (#GP). Algunos atributos son:

1. P: Present flag. 1 if present.

2. DPL: Descriptor Protection Level. Nivel de privilegios del descriptor.

3. D: Size of gate. 1 = 32 bits; 0 = 16 bits.

Un procesador Intel reserva por default las primeras 31 posiciones de la `IDT` para las diferentes excepciones del procesador. Actualmente, el procesador solo utiliza las primeras 21. Inicializamos estas excepciones del procesador a una rutina que guarda el estado del procesador al suceder la primera interrupción y en caso de ser necesario desaloja la tarea actual. También inicializamos otros descriptores para atender otras interrupciones como la del reloj y la del teclado.

Una vez cargada la IDT, se debe remapear el PIC (Programmable Interrupt Controler) para referir a las nuevas interrupciones que agreguemos. Esto se hace con las rutinas de la cátedra `resetear_pic` y luego `habilitar_pic`.

3.4. Otras Interrupciones

Para inicializar otras interrupciones, tenemos que agregar los diferentes descriptores a la IDT, que apuntan a su correspondiente rutina de atención y además tienen los atributos correctos. Recordemos que las rutinas de atención de la interrupción deben ser transparentes a lo que el procesador estaba ejecutando en el momento, por lo que se deben guardar todos los registros utilizados y luego restaurarlos al finalizar la rutina de atención. Además, las interrupciones en general llevan a un escalamiento de privilegios, por lo que los privilegios también deben ser restaurados.

A su vez, la rutina de atención de la interrupción debe indicarle al pic que la interrupción esta siendo atendida, para que otras interrupciones puedan suceder. Esto se hace con la rutina de la cátedra `fin_intr_pic1` .

3.4.1. Reloj

Esta es una interrupción interna, que sucede con cada `tick` del reloj del procesador. La rutina de atención de esta interrupción se encarga de mostrar la animación de un cursor rotando en la esquina inferior derecha de la pantalla, por medio de la función `screen_actualizar_reloj_global` . Luego haremos que llame al Scheduler y haga el switch de tareas.

3.4.2. Teclado

Utilizaremos la rutina de atención del teclado para habilitar las diferentes teclas disponibles a los jugadores. Cuando programábamos esto, notamos que es necesario tomar la tecla presionada desde el controlador del teclado, caso contrario el teclado no vuelve a solicitar una interrupción.

3.4.3. Software

Asignamos a la interrupción `0x46` (70) una rutina que atiende un servicio del sistema.

3.5. Entorno multi-tarea

Para inicializar el entorno multi tearea, es importante definir entradas de `TSS` (Task State Segment) en la GDT. Cada una de estas entradas tienen bits distribuidos como la imágen de abajo, y declaran para una tarea cualquiera dónde se encontrará su Task State Segment correspondiente.

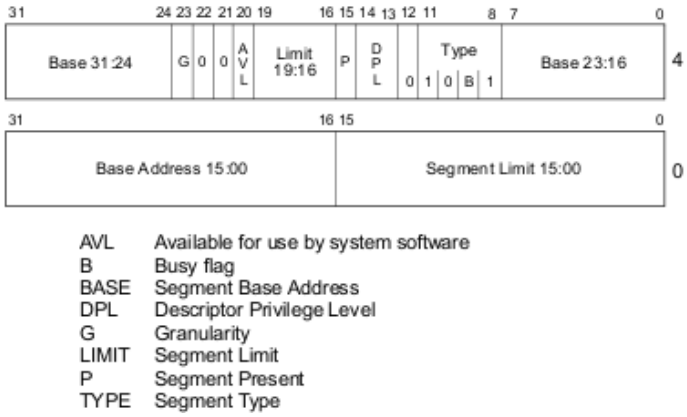


Figura 10: TSS Descriptor

El Task State Segment es una estructura determinada por Intel, con fines de proveer un mecanismo automatizado que cumple con el propósito de guardar todo el contexto de ejecución del programa, facilitando la lógica necesaria para armar un entorno multi-tareas: el procesador se encarga automáticamente de todo el guardado y la carga de registros,

ayudándonos a mantener una lógica más simple y correcta a nivel código. La siguiente figura describe la distribución de bits en un TSS:

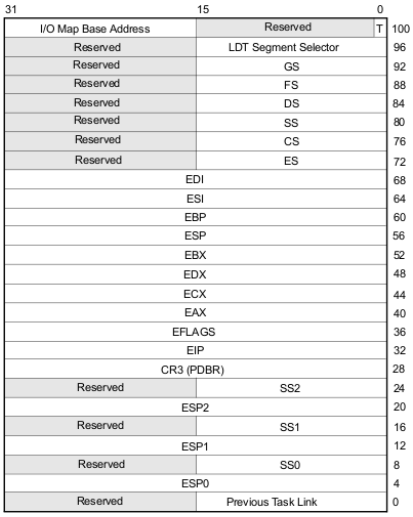


Figura 11: Task State Segment

Estas estructuras deben ser inicializadas con cuidado: es muy importante que cada tarea tenga una pila de nivel 0 bien configurada, una pila de nivel 3, los segmentos de datos y código bien definidos (¡es importante destacar que en el caso de las tareas a nivel usuario hay que declarar los permisos adecuados!), el `eip` , y el `iomap` deshabilitado. Además las `EFLAGS` deben estar puestas en 0x202, ya que sino no tendremos interrupciones habilitadas.

La última pieza del mecanismo es el registro `TR` (Task Register), que contiene todo el tiempo el offset del selector de segmento de TSS correspondiente a la tarea actual. El mismo tiene una parte oculta que cachea el descriptor de segmento correspondiente al selector. Las instrucciones `LTR` y `STR` nos permiten cargar y guardar el Task Register, asumiendo que tenemos `CPL` 0.

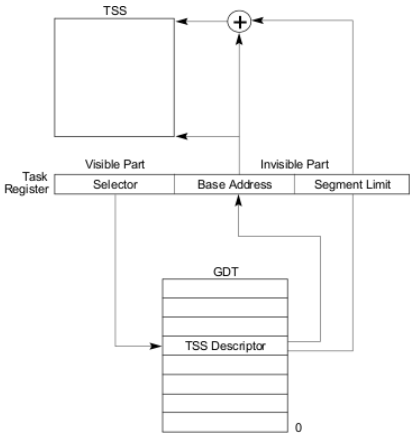


Figura 12: Segmentation & Paging

Para llamar a una tarea, se lleva a cabo un `jmp far` con su respectivo selector de segmento (el offset no se utiliza). El procesador se ocupa de guardar el contexto de la tarea actual en la TSS correspondiente al `TR` , y de cargar el nuevo contexto de ejecución.

En nuestro kernel, definimos un total de 18 tareas: la inicial, la idle, y las 16 tareas de piratas correspondientes a los jugadores. La tarea inicial tiene como propósito guardar el contexto antes de realizar el primer task switch, mientras que la tarea idle funciona como un placeholder cuando no tenemos una tarea de usuario para ejecutar. Por lo tanto, la forma de inicializar multi tasking es cargando el registro `TR` con la tarea inicial, haciendo luego un task switch a la

tarea Idle. Cabe destacar que es importante tener habilitadas las interrupciones antes de saltar a la tarea Idle, ya que sino no podremos ni lanzar piratas, ni ejecutar código de los mismos.

4. Tierra Pirata

4.1. Mapa y manejo de memoria

La memoria se va a distribuir la siguiente manera:

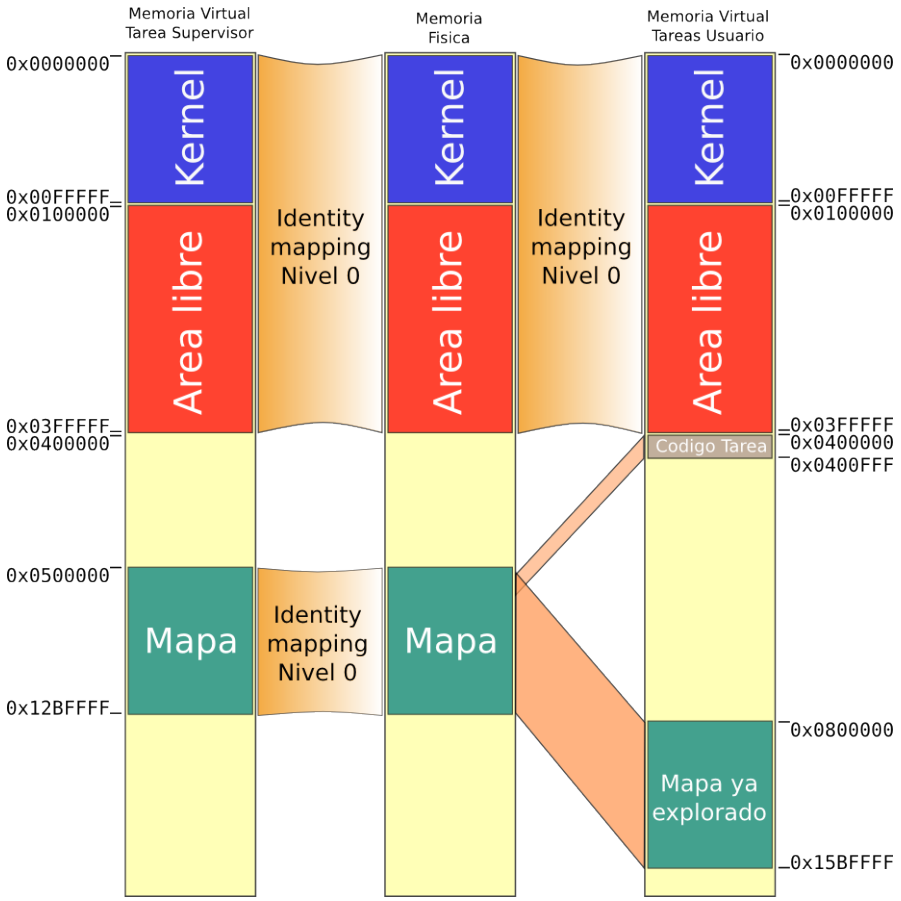


Figura 13: Organización de la Memoria.

La Directory Table del kernel tendrá paginado todo lo útil de la memoria en identity mapping.

Luego cada tarea pirata va a tener su propia Directory Table donde se van a paginar los primeros 4MB de memoria en modo supervisor, luego el código de la tarea, que se encontrara en una posición del mapa, sera mapeado a la dirección virtual 0x400000 en modo user y con derechos de escritura; por ultimo se mapeara el mapa a la posición 0x800000, solo paginando las posiciones ya exploradas por el jugador en modo user con derechos de solo lectura.

Las Directory Tables de las tareas se localizaran dentro del área libre de la memoria, como se ve en la figura 12.

Escribimos funciones de inicialización tanto para la memoria virtual del kernel como para la memoria virtual de los piratas, las siguientes funciones deben ser llamadas en el proceso de inicialización de ambos:

- `void mmu_inicializar_dir_kernel()` : se ocupa de hacer el identity mapping de 4MB del kernel. Además, le mapea todo el mapa en modo escritura, lo que nos permitirá a futuro manejar todo el movimiento de los piratas.
- `int mmu_inicializar_dir_pirata(uint directoryBase, uint pirateCodeBaseSrc, uint pirateCodeBaseDst)` : Se encarga de realizar el identity mapping de los primeros 4MB de memoria física, los cuales incluyen

Hay que tener en cuenta que para poder hacer el correcto funcionamiento del sistema hay que tener lugar asignado para las pilas, tanto de nivel usuario como de nivel 0, ya que al estar ejecutando una tarea, si se recibe una interrupción

tanto interna, de software o externa, el estado del sistema pasara a nivel 0, por lo tanto se necesitara una pila separada de nivel 0.

Por lo tanto la decisión que tomamos tener una pagina (dentro del área libre) por cada tarea pirata, donde colocaremos la pila de nivel 0. Y la pila de nivel usuario se colocara al final de la pagina donde se encuentra su código. En la figura 12 se ve la localización de las pilas de nivel 0 de las tareas pirata.

Con respecto a la necesidad de tener que pedir espacio paginas memoria donde ir colocando las Page Tables para todo el sistema de paginación, ya sean tareas piratas o kernel, vamos a asignándolas en el área libre a medida que sean pedidas, organizándolas como muestra la figura 12.

Figura 14: Organización del área libre.

4.2. Manejo de interrupciones

4.2.1. Reloj

La interrupción de reloj comienza llamando la función `scheduler_tick` , que se ocupa de devolver el índice de la tarea en la `GDT` al que se debe ir. En caso de que se tenga que intercambiar de tarea, la rutina de atención de esta interrupción hace un `jump far` a la nueva tarea.

4.2.2. Teclado

Cuando hay una interrupción de teclado, la tecla que ha sido presionada se codifica con un `scan code` de 8 bits, que se guarda en el controlador de teclado. Estas se leen a través del puerto `0x60` con la instrucción `in al, 0x60`. Una vez guardado el código, se llama a la función de C `isr_keyboard` , a la que se le pasa el código por pila respetando la convención C de 32 bits.

La función `isr_keyboard` luego le asigna diferentes funciones a las teclas `right_shift`, `left_shift` e `y`. Todos los scan codes están definidos en `keyboardcodes.h`.

4.2.3. Syscall

El sistema provee un servicio a las diferentes tareas mediante la interrupción `0x46` . La misma le permite a las tareas usar de forma indirecta las siguientes funciones.

1. `game_syscall_pirata_mover` : Mueve al pirata de una posición a otra. Si el pirata que llamo a esta función es un minero, requiere que la pagina a la que se quiere mover ya este paginada. Al moverse el pirata, también se mueve su código, y se debe también remappear la dirección `0x400000` a la nueva dirección física donde el código se ha movido.

   Si un pirata explorador al moverse encuentra un tesoro, se lanza un pirata minero automáticamente. Al mismo se le debe pasar la posición del tesoro por parámetro. Utilizando la convención C y el stack correspondiente a la nueva tarea, escribimos los parámetros en el stack de la dirección física de su respectivo codigo.

   Cuando programamos esto tuvimos el siguiente problema. La interrupción al mover causaba un escalamiento de privilegios, pero seguíamos manteniendo el `cr3` de la tarea que encontró el tesoro. Por esta razón no podíamos escribir en el stack de la nueva tarea y teníamos un #GP Fault. Esto se debe a que las tareas tienen paginadas la memoria en solo lectura. Para resolver esto, simplemente cambiamos el `cr3` por el del kernel antes de escribir en el stack y luego lo restauramos.

2. `game_syscall_cavar` : Actualiza los puntajes y los atributos del tesoro correspondiente cuando un pirata minero cava. En caso de que se acaben las monedas del tesoro, el pirata se desaloja con la función `game_pirata_exploto`

3. `game_syscall_pirata_posicion` : Devuelve la posición del pirata. La misma se codifica como y << 8 | x en un entero.

El Syscall se llama indirectamente a través de `inline asm`, que se ocupa de pasar los parámetros y llamar a la respectiva función. Al ejecutar esta interrupción hay un escalamiento de privilegios a nivel 0, lo que permite manipular la memoria y ejecutar funciones que no se podrían ejecutar desde una tarea (user), como por ejemplo manipular los directorios de pagina.

4.3. Scheduler

El Scheduler es una función en C que es llamada por la interrupción de teclado. La misma de identificar la tarea que se debe ejecutar actualmente para respetar el siguiente diagrama:

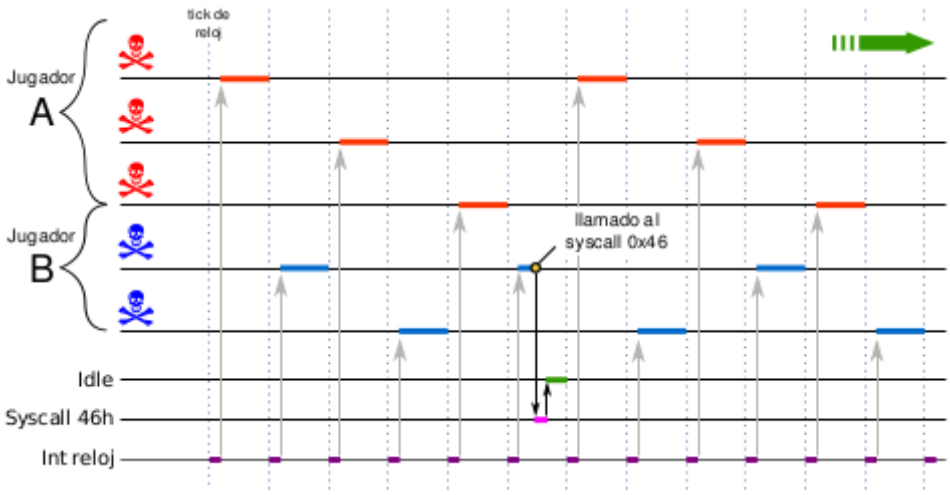


Figura 15: Diagrama de Tareas

El algorítmo de scheduling funciona, a grosso modo (en el código fuente hay comentarios explicandolo más en detalle) manteniendo en todo momento el índice del próximo pirata que debería jugar (tanto para el jugador A como el B), el índice del próximo jugador a ejecutarse, y la cantidad de ticks de timer que lleva (estos ticks se resetean cada `SCHEDULER_TASK_TICKS` interrupciones, y se incrementa cada vez que el timer interrumpe). Entonces, cuando los ticks del timer se vuelven 0, recorre para el jugador correspondiente todas las tareas empezando del índice denotado, hasta alcanzar de vuelta este valor. Si encuentra una tarea existente, la ejecuta, en caso contrario, intenta con el otro jugador. Suponiendo que no encuentre ninguna tarea a ejecutar, termina haciendo el switch a la Idle. Es importante notar que no hacemos task switch si la próxima tarea a ejecutar es igual a la que se está ejecutando, o si el valor de retorno del scheduler no es -1 (esto va a suceder siempre que el contador de ticks no esté en el estado inicial).

Además, cada vez que tenemos una interrupción por software terminamos saltando a la Idle por el resto del quantum de tiempo que le corresponde a la tarea.

4.4. Estructuras

Para mantener cuenta del estado del juego y de cada uno de los jugadores, creamos algunas estructuras definidas en `game.h`. Para saber que posiciones del mapa ya han sido paginadas, cada jugador mantiene un `bit_map` con (`MAPA_ALTO` * `MAPA_ANCHO` / 8) chars. Cada bit corresponde a una pagina del mapa. De esta manera, al lanzar un pirata podemos saber muy fácilmente qué páginas le debemos mapear.