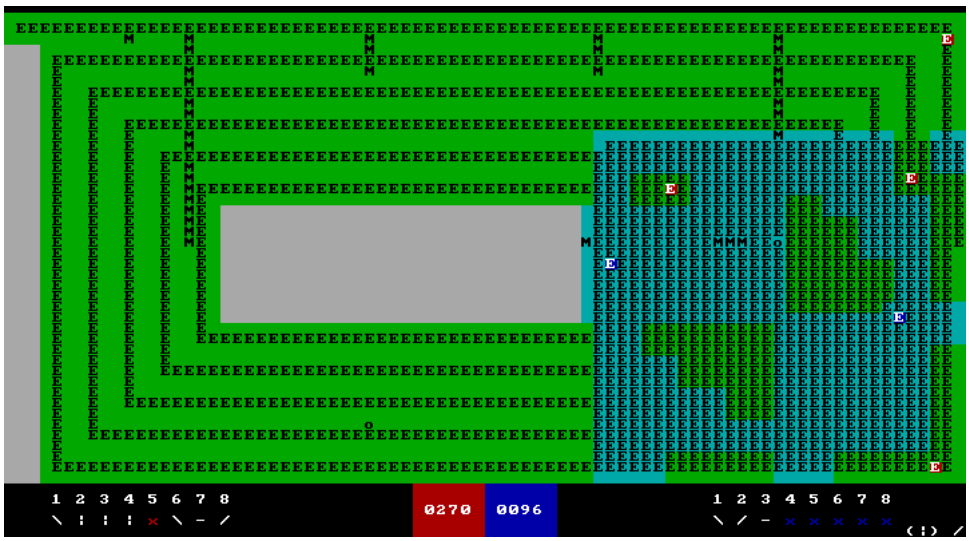
Organización del Computador II

TP3

Tierra Pirata

17 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 inicializacion 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 boooteo, 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 direccion `0x7c00` de un diskette. Esto se copia comenzando en la direccion `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 direccion `0x1200` , ejecutando a partir de la misma.

2. Kernel

El `Kernel` es una parte esencial de los sistemas operativos modernos. Se ocupa de inicializar las diferentes estructuras necesarias para utilizar las diferentes funciones del procesador, como la protección por hardware, la paginación y el manejo de interrupciones.

3. Modo Real

3.1. 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 `Modo Real`, no existe la protección por hardware, por lo que 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.

3.2. 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.

3.3. Global Descriptor Table

Antes de poder pasar a modo protegido, debemos cargar la GDT. La GDT se encarga de asignar diferentes atributos de protección a los segmentos de memoria, para luego poder habilitar la protección por hardware. Esta estructura la armamos como un array de `gdt_entry` en C.

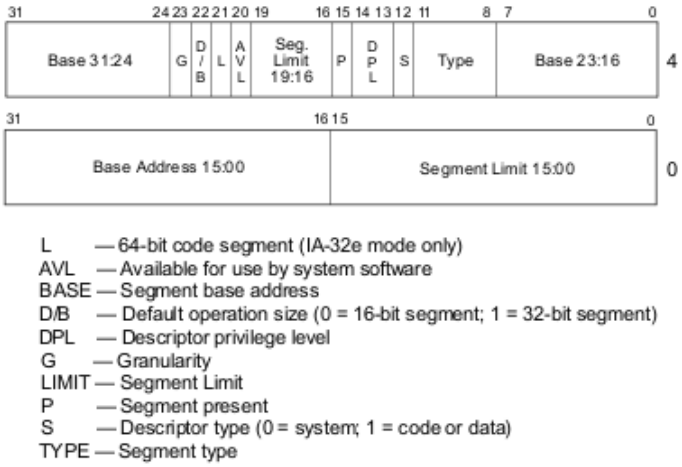


Figura 1: Segment Descriptor

Luego, cargamos la GDT con el comando `lgdt` y el descriptor de la GDT armado desde C (`GDT_DESC`).

En este trabajo utilizaremos el modelo de memoria `flat`. Esto significa que todos los segmentos tendrán como base el address `0x0` . Esto hace que el offset coincida con las direcciones físicas, lo que es sumamente conveniente.

3.4. Pasaje a Modo Protegido

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

Una vez que tenemos todas las estructuras necesarias armadas, hay que hacer un `jump 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.

4. Modo Protegido

4.1. Selector de Segmento

Al entrar a modo protegido, inicializamos los diferentes selectores de segmento. Los mismos tienen el siguiente formato:

Figura 2: Segment Selector

El index corresponde a un descriptor de segmento de 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).

4.2. Niveles de protección

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

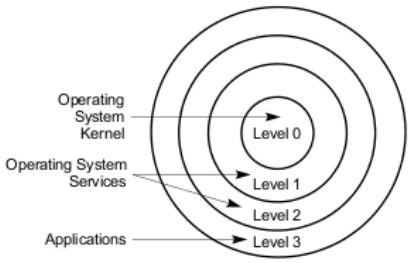


Figura 3: Protection Rings

Si el $max(CPL, RPL) > DPL$ (recordemos que un nivel de protección mayor numerico 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:

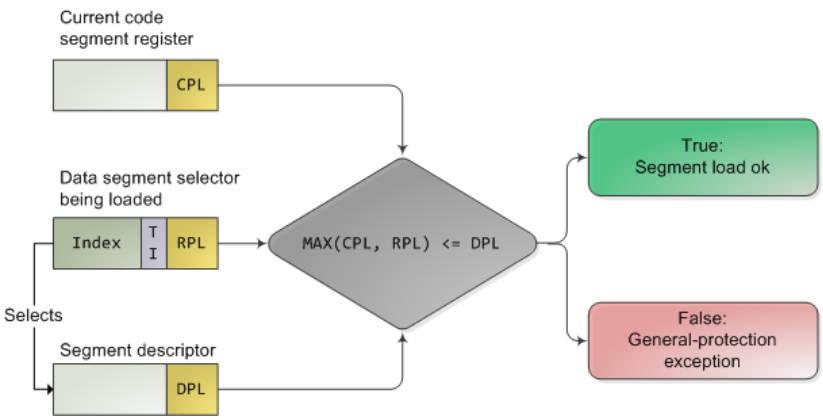


Figura 4: Protection Diagram

4.3. 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
2. Software
3. Internas

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.

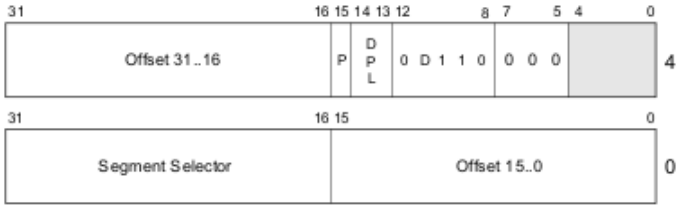


Figura 5: IDT Descriptor

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`.

4.4. 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:

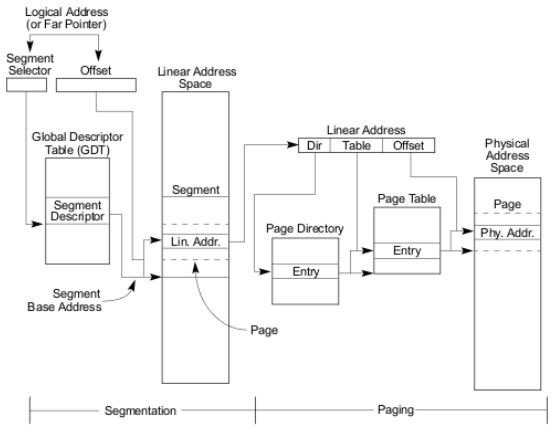


Figura 6: Segmentation & Paging

La paginación nos permite que cada tarea pueda tener su propia `memoria virtual`, mappeando direcciones virtuales a direcciones físicas.

4.4.1. Unidad de Segmentación

La unidad de segmentación se ocupa de pasar desde las *direcciones lógicas* a direcciones lineales. Para ello, utiliza la GDT para identificar el segmento adecuado y luego su respectivo offset. La unidad de protección verifica que el `DPL` es compatible con el `CPL` y el `RPL`.

En modo protegido, los selectores de segmento tienen 16 bits. Los 13 bits mas significativos contienen el indice 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.

4.4.2. 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 7: 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).

4.5. Otras Interrupciones

Para inicializar otras interrupciones, tenemos que agregar los diferentes descriptores a la IDT, que apuntan a su correspondiente rutina de atención y ademas 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. Ademas, 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` .

4.5.1. 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.

4.5.2. Teclado

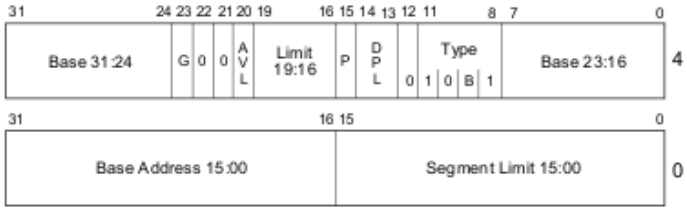
Utilizaremos la rutina de atención del teclado para habilitar las diferentes teclas disponibles a los jugadores. Cuando programábamos esto, notamos que es necesario tomar la tecla presionada desde el controlador del teclado, caso contrario el teclado no vuelve a solicitar una interrupción.

4.5.3. Software

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

4.6. Task State Segment

La `TSS` (Task State Segment) es el espacio de memoria previsto en los procesadores `IA-32` como el espacio de contexto de cada tarea. Este segmento debe tener su respectivo descriptor en la `GDT`.



AVL	Available for use by system software
B	Busy flag
BASE	Segment Base Address
DPL	Descriptor Privilege Level
G	Granularity
LIMIT	Segment Limit
P	Segment Present
TYPE	Segment Type

Figura 8: TSS Descriptor

Este descriptor apunta a un segmento con la siguiente estructura:

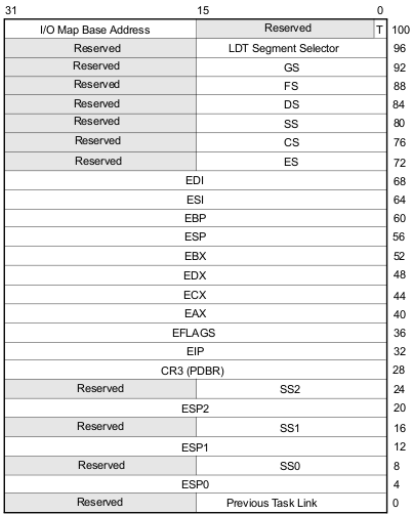


Figura 9: Task State Segment

Estas estructuras deben ser inicializadas con cuidado, con sus respectivos `EIP` , `ESP` , `EBP` , `CR3` e `EFLAGS` entre otros. Por ejemplo, para que las interrupciones esten habilitadas `EFLAGS` debe tener el valor `0x202` .

Ademas, el selector de segmento de la tarea que se esta ejecutando actualmente se debe encontrar en el registro `TR` (Task Register). Este selector tiene 16 bits. Ademas, este registro 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 solo si tenemos `CPL` 0.

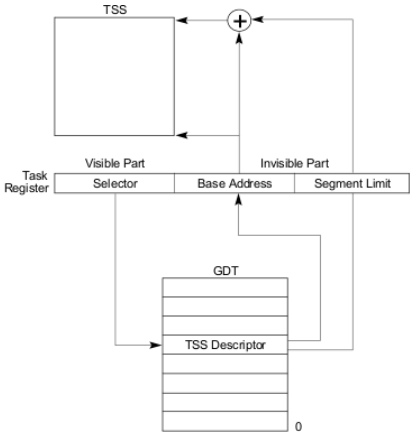


Figura 10: Segmentation & Paging

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

La primera vez que llamamos a una tarea, el registro `TR` no tiene un valor definido. Por esta razon definimos la `tss_inicial` y cambiamos el `TR` antes de saltar a la primera tarea.

5. Tierra Pirata

5.1. Memory Management Functions

Para facilitar el manejo del armado de estructuras para la paginación, se crearon las siguientes funciones en C. Antes de explicar que hace cada función, un comentario. 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. El procesador, al buscar estas estructuras en memoria RAM, requiere que las mismas estén alineadas a 4kb, dado que es el tamaño de pagina que carga en memoria cache.

1. `create_page_table(uint directoryBase, uint directoryEntry, uint physicalAddress, uchar readWrite, uchar userSupervisor)` : Asigna una `page_table` a una tabla de directorios con los atributos pasados por parametro. Al final de la función, se limpia la memoria cache para garantizar que cuando el procesador busca esta pagina, la misma se encuentra actualizada.

2. `delete_page_table(uint directoryBase, uint directoryEntry)` : Borra una tabla de paginas de un directorio de paginas. Esto lo hace simplemente setteando el bit P (present) en cada pagina en 0.

3. `create_page(uint directoryBase, uint directoryEntry, uint tableEntry, uint physicalAddress, uchar readWrite, uchar userSupervisor)` : Crea una pagina en la tabla de paginas de algún directorio.

4. `delete_page(uint directoryBase, uint directoryEntry, uint tableEntry)` : Borra una pagina en la tabla de paginas de algún directorio. Esto lo hace setteando el bit P en 0.

5. `mmap(uint virtualAddress, uint physicalAddress, uint directoryBase, uchar readWrite, uchar userSupervisor)` : Mappea una dirección virtual en una direccion fisica. Para esto, primero se busca la tabla de paginas y la pagina correspondiente a la dirección virtual. Luego se le asigna a esa pagina la dirección física. Esto se hace de la siguiente forma:

   a) A partir de la dirección virtual, se busca la entrada de directorio correspondiente a la misma. Esto se hace dividiendo el virtualAdress por el tamaño direccionable por cada page_directory.$virtualAdress/1024 * 4kb$. Esto es equivalente a $virtualAdress >> 22$.

   b) Buscamos el indice en la entrada de paginas. Esto se calcula dividiendo por el tamaño de pagina e ignorando los bits correspondientes a la entrada de directorio $virtualAdress/4kb$ & $0x3FF$, que es equivalente a $virtualAdress >> 12$ & $0x3FF$.

6. `munmap(uint directoryBase, uint virtualAddress)` : Desmappea la pagina correspondiente a una dirección virtual. Calcula todos los indices necesarios de la misma manera que `mmap`

7. `remap(uint directoryBase, uint virtualAddress, uint physicalAddress)` : Remappea la pagina dada por el *virtualAddress* a la dirección *physicalAddress*.

8. `getPhysVirt(uint directoryBase, uint virtualAddress)` : A partir de un *virtualAddress*, devuelve el *physicalAddress*.

9. `isMapped(uint directoryBase, uint virtualAddress)` : Devuelve si la dirección virtual esta mappeada en memoria.

10. `mmu_inicializar_dir_kernel()` : Inicializa el directorio del kernel. Para ello, hacemos memory mapping sobre el kernel y le asignamos un area libre, todo desde `0x00000000` a `0x003FFFF` .

11. `mmu_inicializar_dir_pirata(uint directoryBase, uint pirateCodeBaseSrc, uint pirateCodeBaseDst)` : Esta función inicializa el directorio de un pirata. Al igual que el Kernel, hacemos memory mapping, aunque en

modo user y en read only. A su vez, mappeamos la pagina donde vamos a poner el codigo del pirata, y copiamos el código del pirata que se encuentra en el Kernel en esta pagina.

12. `mmu_move_codepage(uint src, uint dst, pirata_t *p)` : Mueve la pagina de codigo del pirata desde *src* a *dst*.

    No implementamos la funcion `mmu_inicializar` dado que todo el trabajo lo hace `mmu_inicializar_dir_kernel`.

5.2. Interrupciones

5.2.1. Reloj

La interrupción de reloj comienza llamando la función `scheduler_tick` , que se ocupa de devolver el indice 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.

5.2.2. 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`.

5.2.3. 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.

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.

5.3. 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:

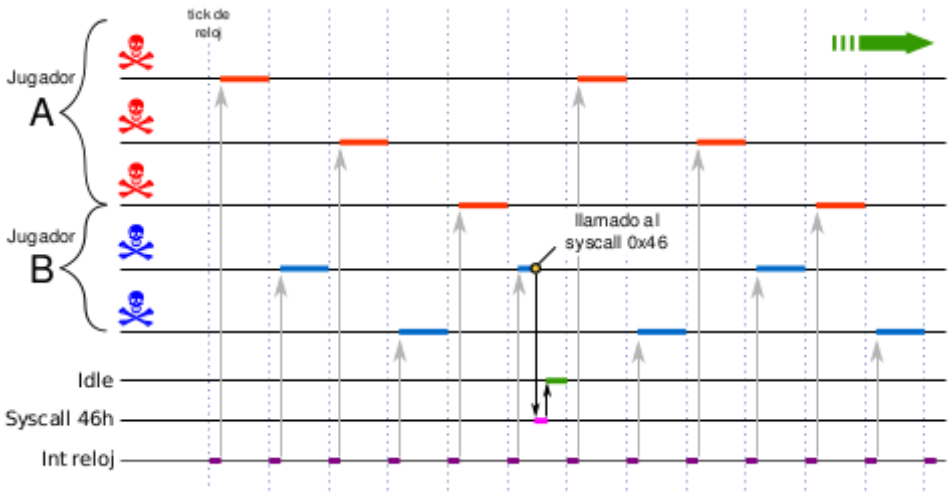


Figura 11: Diagrama de Tareas

Mantiene un contador para saber por cuantos ciclos de clock se ha ejecutado la tarea actual. A su vez, mantiene un flag para identificar a que jugador pertenece la ultima tarea que ha sido ejecutada. Para identificar que tarea ejecutar, itera sobre los piratas del jugador a partir del actual. Una tarea es ejecutada a lo sumo `SCHEDULER_TASK_TICKS` ciclos de clock. Esto esta definido en `defines.h`.

5.4. 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 mover un jugador podemos fácilmente identificar que paginas deben ser mappeadas.

5.5. Funciones Auxiliares

Para facilitar la programación del juego, utilizamos las siguientes funciones auxiliares:

1. `void game_pirata_inicializar(pirata_t *pirata, jugador_t *jugador, uint index, uint id)` :
2. `void game_pirata_erigir(pirata_t *pirata, jugador_t *j, uint tipo)` :
3. `void game_pirata_habilitar_posicion(jugador_t *j, pirata_t *pirata, int x, int y)` :
4. `void game_pirata_exploto(uint id)` :
5. `void game_jugador_setBitMapPos(jugador_t *j, uint x, uint y, uchar val)` :
6. `char game_jugador_getBitMapPos(jugador_t *j, uint x, uint y)` :
7. `void game_pirata_paginarPosMapa (pirata_t *p, int x, int y)` :
8. `void game_jugador_paginarPosMapa_piratasExistentes (jugador_t *j, int x, int y)` :
9. `uint game_xy2addressPhys(int x, int y)` :
10. `uint game_xy2addressVirt(int x, int y)` :
11. `uint game_posicion_valida(int x, int y)` :
12. `int game_jugador_taskAdress(jugador_t *j, pirata_t *p)` :
13. `void game_updateScreen(pirata_t * p, jugador_t * j, int x, int y)` :
14. `uint game_pirateIdtoDirectoryAddress(uint id)` :
15. `pirata_t* id_pirata2pirata(uint pirate_id)` :
16. `uint game_dir2xy(direccion dir, int *x, int *y)` :

5.6. Tareas

5.6.1. Idle

La tarea Idle simplemente tiene un contador que sirve para actualizar el reloj del juego.

5.6.2. Explorador y Minero

Estas tareas simplemente usan los syscalls para moverse por el mapa. No tocamos el código original de la cátedra.