DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico III

**subtitulo del trabajo**

Organización del Computador II
Segundo Cuatrimestre de 2014


| Integrante | LU | Correo electrónico |
|---|---|---|
| Esquivel, Federico Nicolás | 915/12 | alt.juss@gmail.com |
| Hernandez, Nicolás | 122/13 | nicoh22@hotmail.com |
| Kapobel, Rodrigo | 695/12 | rok_35@live.com.ar |





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. Objetivos generales** **3**

**2. Desarrollo** **4**
2.1. Segmentación y activación de modo protegido . . . . . 4
2.2. Paginación . . . . . 4
2.2.1. Directorio de kernel . . . . . 5
2.2.2. Directorios de tareas . . . . . 6
2.3. Interrupciones . . . . . 6
2.3.1. Excepciones . . . . . 6
2.3.2. Interrupción de reloj . . . . . 7
2.3.3. Interrupciones de teclado . . . . . 7
2.3.4. Servicios de sistema . . . . . 7
2.4. Gestión de tareas . . . . . 7
2.5. Scheduler de tareas . . . . . 9
2.5.1. Lanzado y desalojo de tareas . . . . . 10
2.6. Lógica de juego . . . . . 10
2.6.1. Interacción con el usuario . . . . . 10
2.6.2. Interacción con tareas . . . . . 11
2.7. Pantalla . . . . . 11
2.8. Modo debug . . . . . 12

**3. Conclusión** **13**

# 1. Objetivos generales

En este trabajo buscamos activar todos los sistemas para que un procesador trabaje en modo multitarea. Partiendo del estado en que el bootloader nos entrega el control, completaremos las estructuras necesarias para pasar a modo protegido, incorporar paginación, manejar excepciones e interrupciones, lanzar tareas y distribuir los recursos del procesador entre ellas.

# 2. Desarrollo

A lo largo de este trabajo haremos referencia a identificadores y símbolos utilizados en el código. Para ello usaremos este formato: `variable_o_funcion`.

## 2.1. Segmentación y activación de modo protegido

De acuerdo a lo indicado en el enunciado, definimos segmentos de código y datos en la GDT a partir del indice 4. El primer par con privilegios de kernel (DPL 0), con indices 4 y 5 respectivamente, y un segundo par de segmentos con privlegios de usuario (DPL 3) en los indices 6 y 7 de la GDT. Direccionamos los primeros $878$MB de memoria con estos segmentos. Para ello establecimos la base de cada uno en la dirección *0x0* y, para poder representar el límite, usamos granularidad de 4KB. De esta manera, $878$MB se corresponden a $224768$ bloques de 4KB. Como en el limite indicamos el último bloque direccionable, restamos uno a esta cantidad. Luego, el límite calculado para estos segmentos fue *0x36DFF*.

Adicionalmente, definimos un segmento de datos con privilegios de kernel para la memoria de video, basado en la dirección *0xB8000*, en el indice 8 de la GDT. Dado que las dimensiones de la pantalla son $80 \times 50$ caracteres y que para representar cada uno se necesitan dos bytes, el tamaño de este segmento es de $8000$ Bytes. En función de esto definimos el limite del segmento en $7999$, su ultimo byte direccionable.

Una vez que construimos estos segmentos en la GDT, la apuntamos en el registro GDTR y activamos modo protegido. Para ello seteamos el bit PE en CR0 y usamos un jump far para cargar el selector de segmento de código. Tras el salto, configuramos los selectores de segmentos de datos (ds, ss, es, fs y gs), apuntándolos al segmento de datos de kernel que definimos en el índice 5 de la tabla de descriptores. Asimismo, asignamos el valor *0x2700* a los registros esp y ebp para configurar la pila de kernel.

Hecho esto, inicializamos la pantalla. En pos de esto usamos el segmento de datos que definimos para el área de la pantalla, cargando su selector en gs. Luego nos desplazamos por el rango permitido por el segmento, escribiendo *0x7020* en cada par de bytes hasta alcanzar el límite. Este valor corresponde a un espacio con fondo gris.

## 2.2. Paginación

Implementamos paginación de dos niveles, donde construimos las estructuras necesarias en el area libre de memoria, que comienza en la dirección *0x100000*.

Para el manejo de páginas libres utilizamos el puntero `proxima_pagina_libre` que denota la dirección de la próxima página libre, al que inicializamos apuntando al inicio del área libre. Administramos este puntero con la función `mmu_proxima_pagina_fisica_libre`, que devuelve un puntero a la proxima página e incrementa el valor de `proxima_pagina_libre` en 4KB.

A través de estos mecanismos, una vez que ubicamos un directorio de páginas en una página libre, hacemos uso de las funciones `mmu_mapear_pagina` y `mmu_unmapear_pagina` para agregarle y sacarle mapeos de páginas.

En la función `mmu_mapear_pagina` tomamos los siguientes atributos:

| | |
|---|---|
| `virtual` : | Dirección virtual a mapear. |
| `cr3` : | Dirección física del page directory al que queremos agregar una página. |
| `fisica` : | Una dirección física. |
| `attr` : | Atributos a definir en las entradas de la tabla y directorio. |

A partir de estos parámetros, al mapear una página tenemos en cuenta primero si es necesario crear la entrada en el page directory, revisando el bit **P** de la entrada correspondiente a la dirección virtual. De ser así obtenemos una nueva página libre para la tabla de páginas, que inicializamos en 0 para todas sus entradas, y agregamos la entrada en el directorio apuntando a ella usando los atributos de `attr`. Luego de obtener la page table, sea recién creada o previamente existente, asignamos la dirección física alineada a 4KB con los atributos pedidos a la entrada indicada por `virtual`. Hecho esto quedan mapeadas las direcciones `virtual` a `fisica`, y llamamos a la función `tblflush` para limpiar la TLB.

En el contexto de este trabajo solo usamos los bits **P**, **R/W** y **U/S** de los atributos, que coinciden en ambas estructuras. Como en el esquema que utilizamos no necesitamos entradas en tablas con atributos distintos a los definidos en la entrada de directorio correspondiente, usamos los atributos definidos en `attr` para todas las estructuras requeridas en cada mapeo. Asimismo, como veremos en las secciones subsiguientes, el area libre esta mapeada por identity mapping para el kernel y todas las tareas. Debido a esto la dirección física del directorio de páginas se corresponde con su dirección virtual en todo contexto. Esto es relevante porque al mapear páginas no siempre se trabaja con cr3 apuntando al directorio de kernel, por lo que es necesario poder acceder al área libre y las estructuras allí definidas desde cualquier page directory. Al estar mapeada por identity mapping, podemos usar las mismas direcciones para referirnos a los directorios y tablas de páginas desde cualquier contexto.

Para la función `mmu_unmapear_pagina` requerimos una dirección virtual y el directorio en el cual queremos deshacer el mapeo. Con estos datos buscamos la entrada en el directorio. De no estar presente retornamos de la función sin cambios. De lo contrario, obtenemos la entrada correspondiente en la tabla apuntada por ella y anulamos ésta última, dejando todos sus bits en cero. Luego verificamos si hay al menos una entrada presente en la tabla y de no ser así, anulamos también la entrada del page directory. Finalmente, limpiamos la TLB con la función `tblflush`.

A continuación detallamos los procedimientos a través de los cuales construimos los directorios que usamos en este trabajo tanto para el kernel, como para las tareas del juego.

### 2.2.1. Directorio de kernel

Tras pasar a modo protegido e inicializar la pantalla con el procedimiento descrito en la sección anterior (ver 2.1), pasamos a activar paginación a fines de incorporar protección por nivel de privilegio a los accesos a memoria. Para ello, incializamos `proxima_pagina_libre` como mencionamos anteriormente, y creamos un page directory para el kernel.

En el directorio del kernel mapeamos las direcciones *0x000000* a *0x3FFFFF* usando *identity mapping*. Para lograrlo en primer lugar inicializamos los primeros 4KB de memoria a partir de la dirección *0x27000* con ceros, donde luego definimos el directorio de páginas del kernel. Ubicamos la primer página del directorio en la dirección *0x28000*, con atributos de lectura y escritura, nivel de privilegios cero y el bit presente alto. Luego, para mapear el sección de memoria pedida, llenamos la tabla con las direcciones de los primeros 1024 bloques de 4KB de memoria física. De esta manera definimos la última entrada de la tabla con la dirección base 0x3FF000, permitiendo direccionar las siguientes 4096-1 direcciones. Al igual que la definición de la tabla de páginas en el directorio, cada página fue definida con atributos de lectura y escritura, DPL cero y bit presente en uno.

Teniendo armado un directorio de páginas con una tabla de páginas, habilitamos paginación moviendo la dirección del directorio a CR3 y levantando el bit correspondiente en CR0.

### 2.2.2. Directorios de tareas

Controlamos la inicialización de directorios para tareas con la función `mmu_inicializar_dir_tarea`. Para ella requerimos los parámetros `tipo`, que indica el tipo de tarea se esta mapeando, y `fisica`, que define la dirección de memoria física dentro del mapa que debemos mapear. Denotamos el tipo de cada tarea con el enum `task_type`, cuyos valores para tareas sanas, de virus A y B son: `H_type(0)`, `A_type(1)` y `B_type(2)`.

En esta función debemos copiar el código adecuado y construir el directorio y páginas necesarias para que una tarea encuentre sus instrucciones a partir de la dirección virtual *0x8000000* y pueda mapear otra página en la posición en el mapa a la dirección virtual *0x8001000*. Para ello usamos el directorio apuntado por CR3 al momento en que se llamó la función para copiar el código. En él, mapeamos la dirección física con identity mapping temporalmente usando atributos de kernel. A continuación escribimos el código de la tarea en la página recién mapeada. Obtenemos éste último calculando el desplazamiento a partir de la dirección *0x10000*, que resulta del producto entre el tamaño de página y el orden en memoria correspondiente al tipo de tarea. Por ejemplo, para mapear una tarea de tipo `A_type`, cuyo orden en memoria es 1 (siendo 0 la primer tarea, la idle), encontramos su código en la dirección *0x11000*.

Una vez escrito el código, desmapeamos la página usada del directorio actual y construimos el de la tarea. Con este objetivo obtenemos una página del área libre para el directorio, que inicializamos con ceros, y seguimos el mismo procedimiento descrito para el directorio de kernel. Una vez mapeada el área de kernel y el área libre en el directorio de la tarea, vinculamos la posición física indicada a la dirección virtual 0x8000000. Consideramos que en un principio la tarea no esta atacando (mapeando la dirección) a otra tarea, por lo que también vinculamos la posición virtual 0x8001000 a la misma dirección física.

## 2.3. Interrupciones

Para poder atender interrupciones primero debemos llenar la IDT y lo conseguimos llamando a la funcion `idt_inicializar`.

En ella utilizamos un macro en el cual referimos cada entrada $N$ al segmento de código de kernel, con el offset correspondiente su rutina de atención _isr$N$.

Todas las rutinas de atención de interrupciones deben correr con nivel de privilegio cero, por lo que en sus descriptores usamos el selector de segmento correspondiente al indice 4 de la GDT, el segmento de código de kernel. Además, no deben poder ser invocadas por código de usuario, por lo que las definimos con DPL 0. La única excepción es la interrupción *0x66*, que utilizamos para las syscalls a disposición de los usuarios. Por este motivo, su entrada en la IDT especifica DPL 3. Independientemente de su nivel de privilegios, definimos todas las interrupciones con tipo *interrupt*.

### 2.3.1. Excepciones

En un principio, resolvimos el manejo de excepciones definiendo los primeros veinte indices (de 0 a 19) en la tabla de descriptores de interrupción. Para atender excepciones escribimos rutinas que muestran sus mensajes de error por pantalla. En pos de esto, guardamos los mensajes de error correspondientes en cada posición del arreglo `mensajesExcepcion`. Luego, definimos cada _isr$N$ con un macro en el cual obtenemos y mostramos el $N$-ésimo mensaje de `mensajesExcepcion`.

Este comportamiento luego fue revisado para la implementación del desalojo de tareas y el modo debug. En su versión final, cada rutina de excepción muestra la pantalla de debug si esta activado, imprime el mensaje de error y desaloja la tarea causante (ver secciones 2.5.1 y 2.8).

### 2.3.2. Interrupción de reloj

Cuando nos llega un tick de reloj desde el PIC, actualizamos el reloj de sistema situado en la esquina inferior derecha de la pantalla mediante la funcion `proximo_reloj`. Luego pedimos al scheduler el indice de la gdt de la proxima tarea a ejecutar. Logramos esto llamando a la funcion `sched_proximo_indice` (ver seccion 2.5); en caso de devolvernos 0, interpretamos que no debemos cambiar de tarea, sino saltamos usando el indice de gdt obtenido.

Tanto si cambiamos de tarea como si no, se llama a la función `game_tick`, una función que por conveniencia decidimos escribirla en C, que se encarga de mantener la consistencia entre el estado del juego y los datos que se muestran en pantalla. Esto implica volver a pintar el fondo si se sale de modo debug (ver sección 2.8), pintar las vidas y puntos actuales de cada jugador y ademas pintar las tareas, páginas mapeadas y actualizar sus relojes.

### 2.3.3. Interrupciones de teclado

El handler de interrupción de teclado solo lee el puerto del teclado ($0x60$) y se lo pasa a la función `atender_teclado`. Esta decide el curso de accion en base a la tecla presionada.

Las teclas 'w', 'a' , 's' y 'd' mueven al jugador 1 y las teclas 'i', 'j', 'k' y 'l' al jugador 2. Las teclas 'shift' permiten a los jugadores lanzar tareas. Las funciones que implementan estos comportamientos se detallan en la sección 2.6.

Sin embargo estas teclas son ignoradas si se produjo una excepción mientras el sistema se encontraba en modo debug. La tecla 'y' nos permite en todo momento activar o desactivar el modo debug.

### 2.3.4. Servicios de sistema

El handler de esta interrupción pasa como parámetros a la función `manejar_syscall` los registros eax, ebx y ecx. Esta funcion setea el parametro global `en_idle` ya que tanto en el caso de ejecutarse exitosamente la operacion requerida como no, debemos poner a correr la tarea idle por el resto del tiempo asignado a la tarea que llamó a la syscall. Luego chequeamos si la tarea nos paso correctamente el parametro `syscall`, de no ser asi la tarea es desalojada. Caso contrario llamamos a la funcion correspondiente al servicio pedido. Estas funciones son responsables de verificar el/los parametros recibidos. Para mas información sobre las funciones encontradas en `manejar_syscall` ver la sección 2.6

## 2.4. Gestión de tareas

Tras definir y configurar las entradas en la IDT, construimos las estructuras que permiten saltar entre tareas. Para albergar el contexto previo al primer salto, con el objetivo de partir desde un contexto limpio, creamos un TSS vacío, agregamos su descriptor a la GDT en el indice 9. Hecho esto, lo apuntamos en el registro TR. De esta manera le estamos indicando al procesador en que segmento guardar el contexto una vez que saltemos a la primer tarea. Este tss, que denominamos `tss_inicial`, se carga con todos sus registros en 0 ya que no se saltará a el y no se modificará más allá de este primer salto.

La tarea que ejecutaremos incialmente será la Idle. Para poder cambiar a su contexto, definimos su TSS y agregamos el descriptor correspondiente a la GDT en el indice 10. En este segmento definimos los campos de la siguiente manera:

| | |
|---|---|
| `esp0` | Base de stack de kernel |
| `ss0` | Segmento de datos de kernel |
| `cr3` | Directorio de páginas de kernel |
| `eip` | `0x1000` |
| `esp` | Base del stack de kernel |
| `ebp` | Base del stack de kernel |
| Registros prop. gral. | `0` |
| `eflags` | `0x202` |
| `cs` | Segmento de código de kernel |
| Segmentos de datos | Segmento de datos de kernel |

Tanto para estos TSS, como para todos los de tareas lanzadas para el juego, usamos descriptores con nivel de privilegio cero y attributo presente en uno.

Para gestionar el estado de las tareas del juego utilizamos una matriz, `tareasInfo`. Esta tiene dimensiones $3 \times 15$, para acomodar todas las tareas en juego. La primer coordenada se corresponde con los primeros tres valores del enum `task_type`, mientras que la segunda indica el indice o id de tarea. Siempre que se accede guardamos el recaudo de mantenernos solo en los indices válidos para cada tipo de acuerdo a la función `task_type_max` que devuelve la máxima cantidad de tareas de la clase indicada. El tipo de esta matriz es una estructura que llamamos `task_info`. En ésta guardamos los siguientes campos:

| | |
|---|---|
| `alive` : | Flag para denotar si una tarea esta en juego. |
| `owner` : | Dueño de la tarea. Utiliza el tipo enumerado `task_type` |
| `x` e `y` : | Coordenadas en el mapa donde esta ubicada la tarea |
| `mapped_x` y `mapped_y` : | Coordenadas en el mapa de la pagina mapeada por la tarea |
| `gdtIndex` : | Indice de la gdt correspondiente a esta tarea |

Para manejar los TSSs de las tareas del juego optamos por una matriz, llamada `tss_directory`, de la estructura `tss`. Esta matriz tiene las mismas dimensiones que `tareasInfo`, de manera que los indices de cada tss se corresponden con los de la tarea cuyo contexto representa.

Utilizamos también estos indices para definir en que posición de la GDT ubicamos el descriptor de TSS, correspondiendo cada posición válida en la matriz con una ubicación en la GDT. De esta manera, partiendo del indice 11 de la tabla de descriptores globales, los primeros 15 consecutivos corresponden a las tareas sanas, los siguientes 5 a tareas del virus A y los últimos 5 a tareas del virus B. En función de esto, para cada tarea de `tareasInfo` calculamos su posición en la GDT con la siguiente formula: $11 + offset(tipo) + indice$.

Basándonos en estas variables y estructuras, al lanzar una tarea creamos su contexto a partir del tipo de tarea, su indice en `tareasInfo`, su indice en la GDT, y la dirección de memoria física correspondiente a su posición en el mapa. Para ello generamos un directorio de paginas siguiendo el procedimiento descrito en la sección 2.2.2 y obtenemos una página de memoria libre para la pila de nivel 0 de la tarea. Luego cargamos su descriptor en la entrada de GDT indicada y llenamos la tss ubicada en `tss_directory[tipo][indice]` con los siguientes datos:

| | |
|---|---|
| `esp0` | Página de memoria libre |
| `ss0` | Segmento de datos de kernel |
| `cr3` | Directorio de páginas de la tarea |
| `eip` | `0x1000` |
| `esp` | `0x8001000` |
| `ebp` | `0x8001000` |
| Registros prop. gral. | `0` |
| `eflags` | `0x202` |
| `cs` | Segmento de código de usuario |
| Segmentos de datos | Segmento de datos de usuario |

Al inicializar las estructuras del juego y scheduler, dejamos en 0 todos los campos de cada entrada en `tareasInfo`. Hecho esto se lanzan las 15 tareas sanas en posiciones al azar dentro del mapa usando la función `sched_lanzar_tarea`, que detallamos en la próxima sección.

A continuación se detalla cómo implementamos los saltos entre tareas y utilizamos las estructuras que introducimos aquí.

## 2.5. Scheduler de tareas

Una vez terminada la configuración del procesador y construidas las estructuras necesarias, saltamos a la tarea idle. A partir de ese momento controlamos los saltos entre tareas a través de un scheduler. Como se mencionó en la sección 2.3.2, en cada interrupción de reloj el código del scheduler decide si debe saltar, y de ser asi a que tarea, hasta el próximo tick de reloj.

Manejamos esto con la función `sched_proximo_indice`, que retorna el indice en la gdt de la tarea a la que se debe saltar, o cero si no es necesario cambiar de tarea.

Ésta se apoya en las siguientes variables:

| | |
|---|---|
| `en_idle` : | Flag que denota si el último salto fue a la tarea idle. |
| `tareasIndices` : | Arreglo de tres posiciones indexado por el enum `task_type`, que guarda cual fué la última tarea que se ejecutó para cada tipo. Tambien nos referiremos a este como el indice actual para un tipo. |
| `tareasInfo` : | La estructura descrita en la sección anterior (ver 2.4). |
| `currentType` : | Variable de tipo `task_type` que guarda el tipo de la tarea corriendo actualmente. |
| `currentIndex` : | Entero que indica el indice de la tarea correindo actualmente. |

Estas variables se inicializan a la vez que `tareasInfo`, con todos los valores en 0. En primer lugar controlamos si estamos en un estado de interrupción por debug, en cuyo caso no debe cambiarse de tarea. Más detalles sobre este comportamiento se describen en la sección 2.8.

Para buscar el indice en la GDT de la próxima tarea iteramos circularmente tres veces, guardando en la variable `nextType` el tipo que estamos considerando en cada iteración. Ésta variable toma inicialmente el tipo siguiente al actual, siguiendo el orden de tareas sanas luego virus A y por último virus B. Al ciclar tres veces, `nextType` toma el valor del tipo actual en la última vuelta, para manejar el caso en que todas las tareas corriendo son del mismo tipo.

A partir de esto, obtenemos el indice de la última tarea ejecutada para este tipo y partiendo de el, recorremos circularmente todos los indices válidos para el tipo en orden de menor a mayor, sin pasar por el último utilizado.

Si la tarea correspondiente al par tipo-indice con los cuales estamos iterando tiene su flag `alive` alto en `tareasInfo`, bajamos el flag `en_idle`, actualizamos las variables `currentType` y `currentIndex` al tipo e indice que encontramos, así como la entrada correspondiente en `tareasIndices`, y devolvemos el indice en la GDT que almacenamos en `tareasInfo`.

Si al ciclar sobre los indices del tipo `nextType` no encontramos ninguna tarea con indice distinto al actual y la entrada en `tareasInfo[nextType][indiceActual]` tiene su indicador `alive` en 1, estamos en un caso particular. Esto se da cuando hay solo una tarea corriendo para un tipo en particular. En esta situación cambiamos a esa tarea si su tipo es distinto al actual, para lo que indicamos que no estaremos en la tarea idle y actualizamos las variables de tipo e indice actuales antes de devolver el indice de la GDT correspondiente.
Si por el contrario el tipo es el mismo que el actual y el campo `alive` correspondiente es igual a 1, estamos en el caso en que solo hay una tarea corriendo, por lo que se sigue ejecutando y no necesitamos cambiar a otra. En estas circunstancias devolvemos 0, excepto que se estuviera ejecutando la tarea idle. Si así fuera, es necesario saltar a la única tarea existente, para lo que solo bajamos el flag `en_idle` y retornamos el valor indice de la GDT almacenado para esa tarea.

Siguiendo este procedimiento implementamos el comportamiento descrito las consignas de este trabajo para la distribución del tiempo de ejecución entre las tareas. A continuación detallamos los mecanismos que utilizamos para lanzar y dasalojar tareas.

### 2.5.1. Lanzado y desalojo de tareas

Como describimos en la sección anterior, dejando de lado el orden, el factor determinante para definir si se salta o no a una tarea es el campo `alive` en su entrada correspondiente en `tareasInfo`. Vale la pena recordar que, de acuerdo a lo mencionado en la sección 2.4, este campo se deja en 0 al inicializar las estructuras. En consecuencia, para desalojar tareas desarrollamos la función `sched_desalojar_actual`, que baja el flag `alive` de la taera actual (de acuerdo a `currentType` y `currentIndex`), y sube `en_idle`, indicando que pasamos a ejecutar la tarea idle.

Esta función siempre se ejecuta a partir de una interrupción, y queda como responsabilidad del handler realizar el salto a la tarea idle. Dichos handlers pueden ser las rutinas de atención para excepciones o bien la rutina de atención para las syscalls.

Para lanzar tareas usamos la función `sched_lanzar_tareas`, que precisa parámetros `tipo`, `x` e `y`. El primero corresponde al enum `taskType`, mientras que los dos últimos son enteros sin signo de 16 bits, representando las coordenadas en pantalla donde se lanzará la tarea. En primer lugar, controlamos si existe un índice libre para colocar la tarea en `tareasInfo`. Si lo hay, calculamos el indice correspondiente en la GDT en base al tipo e indice de tarea y obtenemos la dirección física asociada a las coordenadas en el mapa. Con estos datos creamos las estructuras necesarias de acuerdo al procedimiento descrito en la sección 2.4. Una vez construidas las estructuras, llenamos la entrada en `tareasInfo` con los campos `alive` en 1, las coordenadas de la tarea y su página mapeada iguales a `x` e `y`, el campo `owner` con `tipo` y `gdtIndex` con el índice calculado.

## 2.6. Lógica de juego

En esta sección describimos el estado del juego que se representa en pantalla. La mayor parte del estado del juego se describe mediante la estructura `jugador`. Esta contiene la posicion en el mapa del jugador, los puntos obtenidos y las vidas restantes.

```
typedef struct jugador_t{
    unsigned int tareas_restantes;
    unsigned short x;
    unsigned short y;
    unsigned short id;
    unsigned short puntos;
} jugador;

jugador jugadores[2];
```

### 2.6.1. Interacción con el usuario

Las siguientes rutinas hacen de interfaz con los eventos de teclado que podemos recibir. El kernel recibe las teclas pulsadas y decide el curso de acción. En caso de la tecla pulsada ser un 'shift', se llama a la función `game_lanzar` con el indice de jugador correspondiente (0 para la izquierda o 1 para la derecha). Ella se ocupa de verificar que el jugador tenga "vidas"(representadas mediante `tareas_restantes`) y que haya lugar en el scheduler (solo se pueden correr 5 tareas por jugador). De ser así, se resta una vida al jugador y se le pide al scheduler que lance una tarea en la posición actual del jugador. En la seccion 2.5 se detalla el procedimiento para lanzar una tarea.

Si las teclas pulsadas fueron las de movimiento, el kernel se encarga de transformarlas a un indice de jugador y una dirección. Representamos las direcciones con el siguiente tipo enumerado:

```
typedef enum direccion_e {IZQ = 0xAAA, DER = 0x441, ARB = 0xA33, ABA = 0x883} direccion;
```

La función `game_mover_cursor`, dado un jugador y una direccion, se encarga de de modificar acordemente la posicion en el mapa de la estructura jugador.

### 2.6.2. Interacción con tareas

Las siguientes son las rutinas que provee nuestro sistema para que las tareas interactúen con el estado del juego. La función `game_soy` recibe un parametro que determina si una tarea es roja, azul o verde. Se setea acoremente el parametro `owner` de la tarea actual. `game_donde` recibe un puntero, donde depositamos las coordenadas $(x, y)$ de la tarea en el mapa. `game_mapear` recibe las coordenadas correspondientes a la página que la tarea desea mapear. Primero se verifica que sean coordenadas validas del mapa y se obtiene la direccion fisica de la página mediante `xytofisica`. Si recibimos coordenadas inválidas, desalojamos a la tarea. Si no le pedimos a `mmu_mapear_pagina` que mapee para la tarea actual la página deseada.

## 2.7. Pantalla

Tratamos de mantener el código que maneja el estado de la pantalla lo más modular posible. Es decir, evitar leer/modificar estructuras ajenas y que las demás secciones de nuestro código no tengan que conocer exactamente como esta compuesta la pantalla.

Logramos esto proveyendo desde `screen.c` funciones que realizan de mediadoras entre la memoria de video y las estructuras del juego. Estas funciones no reciben como parametros ninguna estructura, sino identificadores basicos que den la informacion necesaria. Cabe destacar también que, para mantener consistencia entre los datos mostradas en pantalla y el estado del juego, la mayoría de estas funciones son llamadas en la rutina de atención del reloj (ver 2.3.2).

`screen_pintar_tarea`: Recibe tipo de tarea y el $x$ e $y$ que esta ocupa en el mapa (notar que esto es diferente al $x$ e $y$ que ocupa en la pantalla).

`screen_pintar_jugador`: Recibe el indice de jugador (jugadorA o jugadorB) y su posición en el mapa.

`screen_pintar_mapeo_tarea`: Pinta la página mapeada por una tarea. Recibe posicion en el mapa de la página y tipo de tarea.

`screen_limpiar_posicion`: Limpia una posición del mapa (la pinta de gris claro).

`screen_actualizar_puntos`: Recibe los puntos de los jugadores y los imprime en pantalla.

`screen_actualizar_vidas`: Recibe `tareas_restantes` de ambos jugadores y los imprime.

`screen_actualizar_reloj_tarea`: Recibe el tipo de tarea, el indice (este mismo indice se usa en el scheduler), si esta viva o no la tarea y el dueño actual de la tarea (quien la infecto). Se posee una matriz `clock_State` que indica el ultimo estado de reloj de la tarea, se modifica este estado por el siguiente (este orden definido en el arreglo `clock`) y se lo imprime en pantalla.

`screen_imprimir_log`: Imprime el log correspondiente al modo debug. Recibe un puntero a la pila. De alli se extraen en orden los parametros a imprimir.

## 2.8. Modo debug

De acuerdo a lo solicitado implementamos un modo debug que nos permite visualizar el estado de los registros y la pila al momento que se produce una excepción en una tarea, justo antes de que ésta sea desalojada.

Para ello utilizamos la variable `debugState`, del tipo enumerado `debug_state_type`, que define los estados del mecanismo de debug. Estos estados son `debugEnabled`, que indica que el estado de la próxima excepción será visualizado por pantalla, interrumpiendo momentaneamente el flujo del juego; `debugInterrupted`, el estado al que se pasa cuando ocurre una excepción que debemos mostrar; y `debugDisabled`, que denota el estado normal, en el cual no se detiene la ejecución de tareas al ocurrir una excepción. Este último es el estado inicial del juego.

Cuando se presiona la tecla 'y', usamos la función `debug_switch_state` para controlar el estado al que debe pasar `debugState`. En ella, si el modo debug esta deshabilitado, lo habilitamos. Si por el contrario, el valor de `debugState` es `debugEnabled`, lo deshabilitamos. De estar en estado interrumpido, lo pasamos a habilitado para continuar la ejecución hasta la próxima excepción. En esta situación también indicamos que debemos redibujar la pantalla mostrar correctamente la sección del mapa que ocupa la tabla de visualización de registros.

De esta manera, el usuario solo es capaz de cambiar el estado de `debugState` a habilitado o deshabilitado, dependiendo de su valor previo. Únicamente los handlers de excepciones pueden cambiar el estado a interrumpido. Logramos esto llamando a la función `debug_set_interrupted` al entrar en uno de ellos. Esta función pasa `debugState` a estado interrumpido y devuelve $1$ si previamente el estado era habilitado. De lo contrario devuelve $0$, indicando que no debemos interrumpir la ejecución.

Si el handler recibe un $1$ de `debug_set_interrupted`, encola todos los registros de propósito general, los selectores de segmento, registros de control y las últimas cuatro posiciones de la pila de la tarea. Hecho esto, llama a la función `screen_imprimir_log`, que muestra en pantalla una tabla con toda esta información. Al retornar mostramos el nombre de la excepción en la esquina superior izquierda, desalojamos la tarea causante y saltamos a la tarea idle.

De acuerdo a lo que mencionamos en la sección 2.5, mientras `debugState` tenga el valor `debugInterrupted`, no se producen cambios de tarea en la interrupción de reloj.

# 3. Conclusión

En este trabajo logramos implementar un sistema multitarea con segmentación flat, paginación, manejo de interrupciones y los comportamientos descritos en el enunciado.

A raíz de los problemas que encontramos durante el desarrollo, destacamos la importancia de los niveles de protección (RPL) en los selectores de segmentos en modo protegido y en consecuencia los DPLs de las estructuras del sistema. Nuestro diseño se basa fuertemente en protección de niveles de privilegios para funcionar logrando un sistema robusto en cuestiones de seguridad, bloqueando a las tareas de acceder a secciones de memoria sensibles para el kernel, como pueden ser el scheduler, las secciones de código que manejan syscalls y excepciones.

Destacamos, además, el sistema de paginación como método de organización de memoria. Debido a la facilidad de implementación de los algoritmos de mapeo y desmapeo de memoria (y por lo tanto la comprensión de los mismos), dado que no es necesario llevar cuenta donde se encuentra cada sección libre o qué hay que reacomodar para obtenerlas, porque las mismas se organizan en bloques contiguos al contrario de lo que podría suceder en segmentación.