Organizacion del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico N°3

Grupo enifra

Integrante	LU	Correo electrónico
Serio, Franco	215/15	francoagustinserio@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introduccion

Este trabajo práctico *Tierra Pirata* consiste en aplicar los conceptos de System Programming vistos en clase.

Construimos un sistema mínimo que permite correr hasta 16 tareas concurrentes a nivel de usuario.

El sistema es capaz de capturar cualquier problema que puedan generar las tareas y tomar las acciones necesarias para quitar a la tarea del sistema.

Algunas tareas podrán ser cargadas en el sistema de manera dinamica por medio de uso del teclado.

Con la realizacion de este trabajo, vamos a utilizar los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo enfocados en dos aspectos: el sistema de protección y la ejecución concurrente de tareas.

El informe consta de varias secciones, en las cuales se explica lo realizado para poder hacer funcionar el sistema.

2. Modo Real

Cuando encendemos nuestra computadora, la misma inicia en *Modo Real*. En este modo tenemos una memoria disponible de 1mb, no disponemos de privilegios de ningun tipo, y debemos alinear la memoria a 16 bits para poder operar. Para manejar los distintos tipos de interrupciones, disponemos de rutinas de atencion a las mismas. Podemos acceder a instrucciones de cualquier tipo. Estando en Modo real, lo unico que vamos a hacer, es preparar el sistema para poder pasar a *Modo Protegido*, y realizar todo lo que necesitamos que haga nuestro sistema. Para nuestro sistema, definimos el código de *Modo Real* en el archivo *kernel.asm*:

```
BITS 16
start:
    ; Deshabilitar interrupciones
    cli

    ; Cambiar modo de video a 80 X 50
    mov ax, 0003h
    int 10h ; set mode 03h
    xor bx, bx
    mov ax, 1112h
    int 10h ; load 8x8 font

    ; Imprimir mensaje de bienvenida
    imprimir_texto_mr iniciando_mr_msg,iniciando_mr_len, 0x07, 0, 0
    ; Habilitar A20
    call habilitar_A20
    ; Cargar la GDT
    lgdt [GDT_DESC]

    ; Setear el bit PE del registro CR0
    mov eax, cr0
    or eax, 1
    mov cr0, eax

    ; Saltar a modo protegido
    jmp 0x58:mp
```

Mediante este set de instrucciones estamos preparando nuestro sistema para poder pasar a *Modo Protegido*. Dentro del codigo, estamos habilitando A20 que nos da accesos a direcciones superiores a los 2^{20} bits.
Cargamos la *GDT* (Global Descriptor Table). La *GDT* es una tabla ubicada en memoria que define los descriptores de segmentos de memoria en principio. Luego seteamos el bit *PE* del registro *CR0*.
Una vez activado, vamos a saltar a *Modo Protegido*. Este salto lo realizamos utilizando un far jump al descriptor de segmento de código con nivel de privilegio de sistema. Es el jmp que realizamos en la última linea.

3. Modo Protegido

El *Modo Protegido*, es un poco mas completo que el *Modo Real*, tenemos 4GB de memoria disponible, 4 niveles de protección de memoria y rutinas de atención con privilegios.

3.1. Segmentacion

Para el mecanismo de segmentación necesitamos inicializar la GDT. El primer descriptor de la tabla siempre es *nulo*. También al principio tenemos otros 4 segmentos. 1 de código nivel kernel, 1 de código nivel usuario, 1 de datos nivel usuario y 1 de datos nivel kernel. Los segmentos de user tienen dpl 3 y los de kernel 0. Todos los segmentos tienen el bit D/B en 1 por tratarse de segmentos de 32 bits, y el bit S en 1 por ser de datos/código. A su vez, el límite de los segmentos de la gdt se saca porque en el TP dice que se direccionan los primeros 500MB de memoria. Eso significa que tenemos que usar granularidad de a 4KB de lo contrario no llegamos a los 500MB pedidos. El limite se calcula de hacer (500MB / 4KB) - 1. Esto es porque granularity como dijimos está seteado. El dpl es 3 si se trata de usuario y 0 si se trata de kernel. El tipo es 10 si se trata de de código y 2 si se trata de datos. Hay otro descriptor al comienzo de nuestro sistema para el video. Aquí la base es 0xB8000 a diferencia del cero de las demás entradas. Ésto lo dice la página 2 del enunciado del TP. Todos los segmentos se ponen en presentes.
Nos encargamos de setear los offset de los descriptores de los selectores de segmentos. Luego seteamos la pila del kernel en la dirección 0x27000 que indica el enunciado del TP. Al primer ax le cargamos el índice de la gdt de los datos de nivel 0 y al segundo el indice de la gdt del segmento de video, que son el 9 y el 12 respectivamente.
Todos los índices de la GDT se encuentran en **defines.h**. GDT_count es 31 ya que hay 31 descriptores en la GDT (de los 2^{13} posibles) y los índices más grandes que el 0 (descriptor NULL) están a partir del 8 por restricción en el enunciado de TP.

```
BITS 32
mp:
    ; Establecer selectores de segmentos
    xor eax, eax
    mov ax, 1001000b
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov gs, ax
    mov ax, 1100000b
    mov fs, ax
    ; Establecer la base de la pila
    mov esp, 0x27000
    mov ebp, 0x27000
```

Luego de esto imprimimos en pantalla un mensaje de bienvenida a nuestro sistema e inicializamos la pantalla y el juego.

```
    ; Imprimir mensaje de bienvenida
    imprimir_texto_mp iniciando_mp_msg, iniciando_mp_len, 0x07, 2, 0
    call game_inicializar
    call screen_inicializar
    call screen_pintar_nombre
    call screen_pintar_puntajes
```

Las funciones *screen_inicializar*, *screen_pintar_nombre* estan definidas en **screen.h** y desarrolladas en **screen.c**. Estos métodos se encargan de pintar la pantalla, según lo pedido en el enunciado. La función *game_inicializar* esta definida en **game.c** que inicializa los 2 jugadores y el reloj de cada pirata. Si bien se va a explicar con profundidad más tarde, nuestra GDT tendrá en total, aparte de los 5 segmentos descriptos al comienzo de ésta sección, 1 segmento de video, 1 segmento para la tarea Idle, otro para la tarea Inicial y 1 por cada tarea de cada jugador. La GDT_DESC que cargamos en la sección **Modo Real**es una estructura formada por su longitud y su dirección de memoria. Despues de llenar la GDT, pedimos el sizeof(gdt) - 1 y la referencia de gdt (con &*gdt*) para llenar éstos campos.

3.2. Paginado

Habilitamos la paginación en nuestro sistema de la siguiente manera:

```
; Cargar directorio de paginas
; Inicializar el directorio de paginas
call mmu_inicializar
call mmu_inicializar_dir_kernel
xor eax, eax
mov eax, 0x27000
mov cr3, eax
; Habilitar paginacion
xor eax, eax
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Para mappear y unmapear definimos métodos en el archivo *mmu.c*, para ser mas preciso, mmu_mapear_pagina que devuelve void y recibe una direccion virtual y fisica para el que tiene que mapear, un cr3 que es nuestra dirección de comienzo del page_directory_address y flags de read-write y user-supervisor. Durante el mapeo, lo que se hace es lo siguiente:

- Se saca el índice del page directory shifteando la virtual para la derecha 22 bits.
- Se saca el índice del page table shifteando la virtual 10 bits para la izquierda y despues 22 para la derecha (así nos quedamos con lo del medio).
- Se busca si el page directory en ese índice está presente. Si no lo está se lo pone presente y se pide una pagina libre para el page table. Se establece como base_address del page directory esta nueva página pedida shifteada 12 its para la derecha y en éste nuevo page table se pone como base_address la fisica que nos pasan como parámetro shifteada 12 bits para la derecha.
- Si el page directory en ese índice sí esta presente, sólo se pide el page table al que apunta el base_address de ese page directory. A éste último se le establece como base_address la física que nos pasaron como parámetro shifteada 12 bits para la derecha.
- Luego sólo resta poner los bits correspondientes prendidos, apagados, o los read-write y user-supervisor que nos pasan como parámetro.

Finalmente, mara desmapear una pagina, usamos mmu_unmapear_pagina que toma una dirección virtual y un cr3. Lo que se hace para esto es:

- Se shiftea 22 bits a la derecha la virtual pasada por parámetro para sacar el índice dle page directory.
- Se shiftea 10 bits para la izquierda y 22 para la derecha para quedarse con el índice del page table de la dirección virtual pasada por parámetro.
- Se va al page table de esa dirección virtual sacando lo que apunta la base_address del page directory en el primer índice resuelto y se la pone como no presente.

Siempre que mapeamos o desmapeamos algo de la memoria llamamos a $tlbflush()$ para actualizar el caché de interrupciones llamado Translation Lookaside Buffer o TLB.

Durante el juego, debemos poder paginar de manera dinámica las direcciones de los piratas que se agregan, para poder realizarlo utilizamos el siguiente método:

```
unsigned int mmu_proxima_pagina_fisica_libre()
    unsigned int pagina_libre = proxima_pagina_libre;
    proxima_pagina_libre += PAGE_SIZE;
    return pagina_libre;
```

Los piratas (o tareas) en el juego van a tener que copiar su código a la página en donde están y luego, a medida que se vayan moviendo por el mapa, lo que se mueve es ese código por todo el mapa (que está representado desde la 0x800000 a 0x1520000 y 0x500000 a 0x121FFFF en la física). Para ésto, en **mmu.c** contamos con las funciones memcpy y memmov. La primera usando el cr3 que esta en ejecución, mapea la página destino con ella misma (sólo se va a usar para escribir en memoria). Se copia los datos que se querían copiar a esa página y a continuación se desmapea. La función memmov es similar a la anterior, sólo que en vez de únicamente la página de destina se mapea la página de destina y la página de source, se traslada lo que esta source en destino y se desmapean ambas.

Hay otra función en **mmu.c** que es memcpyPila que lo único que hace es pasarle a la pila de la tarea un valor. Ésta función sólo se usa cuando se quieren pasar los parámetros de x e y a la tarea minero.

La última función importante de **mmu.c** es mmu_inicializar_dir_pirata.
Ésta función primero mapea el kernel, ya que todas las tareas deben tener mapeados el kernel por si hay una interrupción de nivel 0 y se está ejecutando una tarea. Luego se mapean las posiciones donde empieza el pirata y sus 8 posiciones circundantes. Para esto tenemos la ayuda de dos funciones; pos2mapVir y pos2mapFis que dados un x y un y devuelve la dirección de memoria de la página de esas coordenadas. Ésto último, a diferencia del kernel, se mapea con nivel de usuario (es decir, 1).

3.3. Manejo de Interrupciones

Para poder atender los distintos tipos de interrupciones, definimos las tareas de atención en *isr.asm* Definimos una rutina para atender las interrupciones del reloj:

```
global _isr32
_isr32:
    ; PRESERVAR REGISTROS
    pushad
    call fin_intr_pic1
    cmp byte [modoDebugPantalla], 1
    je .fin
    call sched_tick
    str cx
    cmp ax, cx
    je .fin
    mov [sched_tarea_selector], ax
    jmp far [sched_tarea_offset]
    .fin:
    ; RESTAURAR REGISTROS
    popad
    iret
```

Luego, para ateneder las interrupciones correspondientes al teclado lo hacemos de la siguiente manera:

```
global_isr33
    _isr33:
    pushad
    call fin_intr_pic1
    xor ax, ax
    in al, 0x60
    push eax
    call game_atender_teclado
    pop eax
    popad
    iret
```

El método *game_atender_teclado* esta definido en nuestro archivo *game.c*. Este método lo que hace, es imprimir en el rincon derecho superior de la pantalla la tecla que se presionó. Cuenta con un switch, que evalua el caso de cada tecla posible, y en base a esto imprime lo que corresponde.

La interrupción del software la realizamos con

```
global _isr70
_isr70:
    cmp eax, 3
    je .posicion
    pushad
    push ecx
    push eax
    call game_syscall_manejar
    call sched_intercambiar_por_idle
    mov ax, 13<<3
    mov [sched_tarea_selector], ax
    jmp far [sched_tarea_offset]
    pop eax
    pop ecx
    jmp .fin
.posicion:
    mov edi, eax
    push ecx
    push edi
    call game_syscall_manejar
    pop ecx
    pop edi
    jmp .fin2
.fin:
    popad
.fin2:
    iret
```

Ésto compara si la interrupción es por la función posición, en cuyo caso se hace aparte para poder devolver el resultado en eax. En el otro caso, llamamos a manejar e intercambiamos por idle. Después hacemos el jmp far para cambiar la tarea.

Para atender las interrupciones de excepciones que describe el ejercicio 2 del TP, definimos una macro:

```
% macro ISR 1
global _isr%1
_isr%1:
    mov eax, %1
    imprimir_texto_mp desc_%1, desc_len_%1, 0x07, 3, 0
    jmp $
```

De esta manera, podemos definir el mensaje correspondiente para cada interrupcion, y utilizando la macro, no tenemos que repetir el código, ya que para todas las interrupciones va a trabajar de la misma manera. Va a imprimir en pantalla el nombre de la interrupción que acaba de ocurrir. Por ejemplo, cuando queremos identificar la interrupción de "Divide Error", definimos el mensaje de la siguiente maner:

```
desc_0 db 'Divide Error'
desc_len_0 equ $ - desc_0
```

De esta misma manera definimos todos los mensajes correspondientes a las interrupciones, en nuestro archivo *isr.asm*.
En el kernel.asm las interrupciones están representadas por el siguiente código:

```
call idt_inicializar
lidt[IDT_DESC]
call resetear_pic
call habilitar_pic
```

Donde lo que se hace es inicializar las entries de la idt, inclusive la entry de la syscall que necesita otro dpl (en este caso 3, de usuario) a diferencia del 0 que es kernel. Por eso hay IDT_entry y IDT_entry_syscall en **idt.c**. La IDT se

carga con la instrucción lidt con la dirección de la idt y se resetea y habilita el Programmable Interrupt Controller, que es el mismo al cual le avisamos en las isr's cuando se atiende cada una de las interrupciones. La linea *sti* activa las interrupciones (set interrupts). Ésto lo hacemos después de cargar la tarea inicial.

3.4. Tareas

Completamos las TSS utilizando código c, en el archivo *tss.c*. Se necesita un idle que va a hacer como pivot en todo cambio de tareas y una inicial para que el primer context switch pueda tirar sus datos en algún lado, por eso no es importante qué tiene y se llena toda con ceros (o basura).

Debemos agregar los *descriptores de TSS* correspondientes a la tarea *idle* y la tarea *inicial* a nuestra *GDT*, para poder realizar esto, definimos un método llamado *tss_ agregar_ a_ gdt*

Luego necesitamos completar la TSS correspondiente a cada pirata.

En el kernel.asm que es lo que se corre en bochs, las tareas estan representadas por el siguiente código:

```
call tss_inicializar
call tss_agregar_a_gdt
```

Esto lo que hace es inicializar los valores de las tss_idle y tss_inicial que están declaradas en **tss.h** y agregar sus respectivos descriptores a la GDT. El código

```
mov ax, 0x70
ltr ax
```

carga la tarea inicial y

```
jmp 0x68:0
```

cambia a la tarea idle. Ésta tarea tiene como cr3, base de la pila y stack el 0x27000, el eip de 0x16000 que es donde está el código de la tarea.

Cada pirata es una tarea distinta y tienen cada uno un cr3 distinto. Cuando se lanza una tarea lo primero que se hace es agregar el descriptor de tss de ese pirata a la GDT y despues se completa la tss. Ésto se hace con los métodos tss_agregar_pirata_a_gdt y completarTssPirata, respectivamente. Ambos estan en el archivo **tss.c** Las entradas de los descriptores de TSS en la GDT tienen como límite 0x0067 porque es el tamaño de la TSS. Se corren en nivel usuario, por eso su dpl es 3, el tipo según el manual de Intel tiene que ser 9 y la base apunta a la dirección de memoria de su correspondiente TSS. Para esto tenemos dos arreglos de TSS, uno para el jugador A y otro para el jugador B de size 8 (porque cada uno puede tener como máximo 8 piratas en ejecución). Por supuesto, todas estas entradas en la GDT se marcan como presente. Cuando se completa una tss, esto es con el método completarTssPirata lo que se hace es lo siguiente:

- Pido una pagina de memoria libre que voy a utilizar para la pila de nivel cero de la tarea en cuestión. Le sumo 4KB (el tamaño de una página) para que el stack pointer apunte al tope de la pila.
- Tomo el puntero a la tss a llenar.
- Pongo como stack pointer de la pila de nivel 0 a la pagina descripta en el punto uno de ésta enumeración.
- El stack segment de nivel kernel es el índice del segmento de datos de nivel kernel en la GDT shifteado 3 para sacar el table indicator y el RPL.
- El eip va a ser el 0x400000 (requerimiento del TP, es la virtual a donde despues se va a copiar el código de la tarea).
- Los eflags siempre que están activadas las interrupciones Intel dice que hay que ponerlos en 0x202.
- El es, ss, ds, fs y gs todos van a tener el índice en la GDT del segmento de datos de usuario (shifteados también 3 para sacar el TI y el RPL).
- El cs va a tener el índice de la gdt correspondiente a el segmento de código nivel user (shifteado 3 para sacar el TI y RPL).

- Para todos estos segmentos hacemos un OR con 3 porque el DPL es 3 (user).
- El manual de Intel indica que el iomap hay que ponerlo todo en 1.
- El esp y ebp tendrán como valor el 401000, ya que según el enunciado del TP comparten el mismo espacio que la tarea y crecen desde la base. Se les resta 12 porque también en el enunciado se dice que las tareas esperan tener apilados sus 2 argumentos y una dirección de retorno y como estamos en 32 bits cada uno de esos 3 datos son 4 bytes.
- El cr3 se usa la función mmu_inicializar_dir_pirata que toma como parámetros al jugador de esa tarea y a la referencia de la tarea.

En el código **defines.h** hay una constante que es EMPIEZAN_TSS que sirve para saber a partir de qué índice en la GDT, las entradas pertenecen a las tareas. En nuestro caso es 15.

3.5. Scheduler

Decidimos representar al *Scheduler* de la siguiente manera:

```
int tareaActualA;
int tareaActualB;
uint proximaTareaA; //indice 0-7
uint proximaTareaB; //indice 0-7
uchar turnoPirata; //0 A, 1 B
uchar turnoPirataActual; //0 A, 1 B
uchar estaEnIdle; // 0 NO, 1 SI
uint modoDebugPantalla;
uint modoDebugActivado;
int proxTareaAMuerta;
int proxTareaBMuerta;
```

Para inicializar estos valores, definimos el siguiente método:

```
void sched_inicializar(){
    turnoPirata = 0;
    proximaTareaA = 0;
    estaEnIdle = 1;
    proximaTareaB = 0;
    modoDebugActivado = 1;
    tareaActualA = 0;
    tareaActualB = 0;
    turnoPirataActual = 0;
    proxTareaAMuerta = 0;
    proxTareaBMuerta = 0;
    modoDebugPantalla = 0;
}
```

El sistema de context-switch para las tareas es el denominado round-robin, es decir, le toca un ciclo de reloj a cada jugador. El jugador que empieza el es A, la primera tarea que empieza a correr es la idle por eso está seteado. Ambos jugadores tienen como primer tarea a correr la 0 o sea la primera. El modoDebugActivado está seteado porque el enunciado dice que la primer excepción del procesador ya tiene que lanzar la pantalla y hay dos flags porque uno indica si está la pantalla debug y otra si está el debug activado. proximaTareaA y proximaTareaB indican las proximas tarea a correr de cada jugador. Por último, la proxTareaAMuerta y proxTareaBMuerta indican cuales son los indices de los piratas de cada jugador que van a ser usados en caso de que se lance un pirata verde o azul, respectivamente.

Para atender a la proxima tarea, vamos a reutilizar la función que cambia las tareas de acuerdo al tick del clock, utilizamos a la función:

```
unsigned int sched_proxima_a_ejecutar(){
    return sched_tick();
}
```

Luego, para realizar lo correspondiente al tick del clock usamos sched_tick, que a su vez llama a game_tick que sólo actualiza todos los relojes, en concordancia a lo que pide el ejercicio

Por último, otros métodos que definimos en nuestro *sched.c*

```
void sched_intercambiar_por_idle(){
    estaEnIdle = 1;
}
void sched_nointercambiar_por_idle(){
    estaEnIdle = 0;
}
void sched_toggle_debug(){
    if (modoDebugPantalla) {
        game_modoDebug_close();
        modoDebugPantalla = FALSE;
    } else {
        if (modoDebugActivado) {
            modoDebugActivado = TRUE;
        } else {
            modoDebugActivado = FALSE;
        }
    }
}
```

Éste último método se llama cuando se aprieta la tecla "*y*". Lo que hace es preguntar si está la pantalla del debug en el juego, en este caso la cierra y vuelve al juego, poniendo el flag en FALSE, que esta definido como cero. Si no está la pantalla cheque si ya se apretó la tecla "*y*" y el modoDebugActivado está activado o no, haciendo un simple toggle de éste último valor.
A su vez, el scheduler se tiene que encargar de sacar una tarea cuando ésta muere y agregar una tarea cuando se lanza. Ésto se hace con sched_sacar y sched_agregar, respectivamente. Lo que hace el primero es fijarse cual es la próxima tarea muerta del jugador en cuestión y actualiza si la tarea que se murió viene antes. El sched_agregar se fija cual esta muerto de las tareas del jugador y actualiza su variable próxima tarea muerta.
El scheduler si inicialize en el kernel.asm cuando se hace

```
call sched_inicializar
```

3.6. modoDebug

Para el modo debug usamos 2 flags que están en la estructura del scheduler, modoDebugActivado y modoDebug-Pantalla.

```
pushad
call game_pirata_explotoisr
cmp byte [modoDebugActivado], 1
jne .fin
mov eax, cr0
push eax
mov eax, cr2
push eax
mov eax, cr3
push eax
mov eax, cr4
push eax
push cs ; push segmentos
push ds
push es
push fs
push gs
push ss
```

```
    push esp ; push array con toda la info de los registros
    call game_modoDebug_open
.fin:
    jmp 0x68:0 ;jumpeo a la idle
```

Cuando se produce una excepción del procesador se va a la rutina de excepción de interrupciones de arriba. Ésta rutina lo que hace es:

- Pushea los registros de proposito general.
- Llama a game_pirata_explotoisr que mata a la tarea en cuestión, es decir la saca del scheduler, la borra de la pantalla, reestablece el clock del pirata en cero, etc.
- Pushea los registros de control a la pila.
- Pushea los segmentos a la pila.
- Pushea el esp, que es lo que se le pasa como parámetro a la función game_modoDebug_open.
- Si el flag modoDebugActivado está apagado no se hace nada de ésto y simplemente se va a la tarea Idle, haciendo un jmp far a su selector de segmento en la GDT.

Las funciones game_modoDebug_open y su contraparte game_modoDebug_close están ambas definidas en el archivo **game.c**. La primera toma el puntero pasado por parámetro para ir para atras en la pila y guardar todos los registros y valores necesarios para la pantalla debug. También guarda el estado de la pantalla del juego en el buffer que denominamos como mapa_backup con un puntero al egmento de video antes de popear la pantalla debug y pone el flag modoPantallaDebug en 1. La segunda sólo reestablece la pantalla del juego utilizando el puntero al segmento de video y el buffer mapa_backup.
En el archivo **defines.h** hay una serie de constantes que nos sirven a la hora de implementar el modoDebug. Éstas son:

- DEBUG_REGISTROS_X: La coordenada X del borde superior izquierdo del rectángulo imaginario donde se empiezan a listar los datos.
- DEBUG_REGISTROS_Y: La coordenada Y del borde superior izquierdo del rectángulo imaginario donde se empiezan a listar los datos.
- DEBUG_CORNER_X: La coordenada X del borde superior izquierdo de la pantalla debug.
- DEBUG_CORNER_Y: La coordenada Y del borde superior izquierdo de la pantalla debug.
- DEBUG_WIDTH: El ancho de la pantalla debug.
- DEBUG_HEIGHT: El alto de la pantalla debug.

Por último, por afuera del modoDebug, se implementó un nuevo syscall, con un tipo 4 para que sea diferente a los tipos de los syscalls provistos por la cátedra, que imprime en pantalla el valor que se le pasa. Éste syscall resultó muy útil cuando se quería inspeccionar algún dato o valor desde los códigos de las tareas. Es el último método definido en el archivo **syscall.h**

4. Juego

Antes de terminar el informe quería hablar un poco de la dinámica del juego en sí. El juego *TierraPirata* es bastante simple en su funcionamiento. Cada jugador puede lanzar un máximo de 8 piratas. Los exploradores son los encargados de ir por el mapa y lanzar mineros cuando se descubran tesoros. El juego se termina cuando se acaba el tiempo definido como MAX_SIN_CAMBIOS en el archivo **game.c** sin que los dos jugadores hayan hecho ningún movimiento o cuando todos los tesoros fueron descubiertos y cavados. Resulta ganador el jugador con más puntos al momento de terminar el juego. Cada punto se le va restando a la tercer tupla del arreglo botines. La posición inicial de los piratas del jugador A es la (1, 2) porque se tomó que la primera linea es una línea negra que no forma parte del mapa del juego. La posicion inicial de los piratas del jugador B es (78, 43). Los colores del jugador A es el verde y del jugador B es el azul. Éstos colores son los que se van a usar cuando los piratas exploradores de cada jugador vayan moviéndose por el mapa. Las posiciones que fueron exploradas por piratas del jugador A como del jugador B serán pintadas de cyan. El explorador se denotará con una E y el minero con una M. Para todo lo demás que concierne a la pantalla se siguieron las recomendaciones de la cátedra.

El juego tiene 2 estructuras definidas. La estructura pirata, que consta de:

- index: El indice del pirata del jugador. Tiene un rango de 0-7.
- jugador: Puntero al jugador al cual pertenece ese pirata.
- id: Id unequívoco del pirata. Tiene un rango de 0-15.
- posicionX: Coordenada x de la posición actual del pirata.
- posicionY: Coordenada x de la posición actual del pirata.
- tipo: Tipo del pirata en cuestion. Puede ser minero o explorador. Usa el enumerado tipo pirata definido mas arriba en **game.h**.
- reloj: Estado del reloj del pirata. Tiene un rango de 0-3.
- vivoMuerto: Flag para saber si el pirata esta vivo (1) o muerto (0).
- posicionXObjetivo: Coordenada X del objetivo del pirata. Sólo usado para los mineros. De lo contrario vale -1.
- posicionYObjetivo: Coordenada Y del objetivo del pirata. Sólo usado para los mineros. De lo contrario vale -1.

y la estructura jugador que tiene las siguientes propiedades:

- index: indice del jugador. Es 0 si se trata del jugador A y 1 del B.
- piratas[MAX_CANT_PIRATAS_VIVOS]: Arreglo de piratas. Son los 8 piratas de cada jugador.
- puntaje: puntaje del jugador.
- piratasRestantes: Piratas que le quedan al jugador lanzar.
- minerosPendientes: Mineros que están pendientes lanzar por ese jugador.
- posicionesXYVistas[80][45]: matriz de booleanos que representa el mapa. La posicion x,y fue vista por ese jugador si y sólo si posicionesXYVistas[x][y] == 1.
- puertoX: Coordenada X de donde salen los piratas del jugador
- puertoY: Coordenada Y de donde salen los piratas del jugador.
- colorJug: Color del jugador. Verde si es el jugador A y azul si es el jugador B. Utiliza los valores C_BG_GREEN y C_BG_BLUE, respectivamente, que se proveen en el archivo **colors.h**

En **screen.c** tenemos varias funciones que complementan el juego. Las principales son:

- screen_pintar_nombre: Pinta el nombre del grupo en el márgen superior derecho de la pantalla.
- screen_pintar_puntajes: Pinta por primera vez los puntajes de los jugadores.
- screen_stop_game_show_winner: Muestra el jugador que ganó el juego.
- screen_actualizar_puntajes: Actualiza los puntajes de los jugadores en la pantalla cada vez que algún jugador hace un punto.
- screen_pintar_reloj_piratas: Pinta los relojes de todos los piratas del juego.
- screen_actualizar_reloj_global: Pinta el reloj de la tarea Idle y el reloj global, así como llamar a la anterior.
- screen_actualizar_posicion_mapa: Actualiza la posición de los piratas en la pantalla.

Por último, para hacer que se corte el juego cuando se llega a un cierto tiempo sin ningún cambio en ninguno de los dos jugadores, se estableció un variable global en **game.h** MAX_SIN_CAMBIOS. En cada ciclo de clock se incrementa otra variable, definida como contador_de_tiempo, que, en caso de ser igual a la primera, se termina el juego.

5. Conclusion

Como conclusión, se pudo implementar un sistema que corra 16 tareas concurrentemente, con funciones únicas a cada tarea, que esté atento a interrupciones, tanto externas como internas. También se pudo comprender cómo funciona el direccionamiento de la memoria. A su vez las tareas se pudieron intercambiar satisfactoriamente. Otros puntos que se pudieron implementar son un sistema básico de I/O en C para la pantalla y una dinámica de juego aceptable. Con este trabajo se pudo entender mejor como funciona un sistema complejo con varias tareas, las funcionalidades y la arquitectura de los procesadores Intel, que entiendo será un tema básico a saber en la próxima materia de Sistemas Operativos.