DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

El Kernel contraataca

Organización del Computador II

Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Alejandro Mignanelli	609/11	minga_titere@hotmail.com
Facundo Barañao	480/11	facundo_732@hotmail.com
Ian Sabarros	661/11	iansden@hotmail.com

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

Resumen

En el presente trabajo se describe el desarrollo del Kernel para una arquitectura intel de 32-bits, así como el manejo de paginación, manejo de tareas, interrupciones y todo lo referente al manejo de un pequeño sistema operativo.

Índice

1. Objetivos generales

El objetivo de este trabajo práctico es, partiendo de un procesador intel de 32-bits, generar un kernel capaz de gestionar memoria entre diferentes tareas, correrlas de manera concurrente, y resolver las diferentes problemáticas que puedan surgir al momento de ejecución.

Para ello utilizaremos las diversas herramientas que intel pone a nuestra disposición en modo protegido: Usaremos segmentación y paginación para controlar el privilegio con el que las tareas se ejecutarán, además de limitar lo que las tareas puedan 'ver' con un mapeo parcial de la memoria. Utilizaremos interrupciones del procesador que permitirán, tanto reaccionar de manera apropiada cuando se produzca un error en tiempo de ejecución, obtener input del teclado y gestionar un task manager que nos permita ejecutar tareas de manera concurrente.

En el presente informe, se detallará de manera más elaborada todo lo hecho para lograr el objetivo del trabajo práctico, así como cualquier decisión que se haya tomado en el código a tales fines. Para su mejor entendimiento, este informe se dividirá en ejercicios, que son pequeñas partes del trabajo, y todos juntos conforman al trabajo práctico en sí.

2. Ejercicio 1: GDT Basica

2.1. GDT básica y pasaje a modo protegido

a) Completamos la GDT de la siguiente manera: El primer descriptor se completa con ceros, ya que este debe ser nulo siempre; las siguientes 7 entradas no se usan debido a que se consideran ocupadas; la octava y la novena entrada corresponden a un descriptor de código de nivel 0 y 3 respectivamente; las dos siguientes son descriptores de datos de nivel 0 y 3 respectivamente. Para un mejor entendimiento observar la figura "GDT".

   Los descriptores de segmento de código llevan un 8 en su campo type (EXECUTE ONLY) mientras que los de datos llevan un 2 (READ/WRITE). Para lograr direccionar los primeros 500MB de memoria, establecemos el límite en `0x1F3FF` y la granularidad en 1.

b) En cuanto al pasaje a modo protegido, en primera instancia se deshabilitan las interrupciones (`cli`) y se habilita el pin A20 para poder acceder a direcciones superiores al primer MB. Luego se carga la GDT con la instrucción `lgdt` y la estructura `gdt_descriptor` definida en `gdt.h`. Lo siguiente es setear el bit PE del registro CR0 y hacer el salto mediante la instrucción `jmp` a la dirección referenciada por la etiqueta `modo_protegido` en la entrada 8 de la GDT, con nivel de privilegio 0. Finalmente, se cargan los registros de segmento (DS, ES, GS y SS) y se establece la base de la pila de kernel en `0x27000`.

c) Asignamos la entrada 12 de la GDT para el área de pantalla. Se establece la base en `0xB8000` y su límite en `0x1000`, el campo type se define como 2 (READ/WRITE) y el bit S en 1. Una vez hecho esto, movemos el selector de esta entrada al registro de segmento FS.

d) Utilizamos la funcion `inic_video` para limpiar la pantalla e inicializarla como muestra la figura 1. Hacemos uso de la estructura que se encuentra en `screen.h` para ir asignando los distintos caracteres y sus atributos (color de caracter y su color de fondo). Para esto usamos la dirección física del buffer de video `0xB8000`.

Figura 1: Estado inicial de la pantalla

3. Ejercicio 2: IDT Basica

a) Completamos la IDT con las primeras 19 entradas, que corresponden a diferentes rutinas de excepciones del procesador. Las excepciones reservadas por intel no son consideradas en este punto. Cargamos las entradas como *Interrupt Gate*; bit de presente en 1, DPL 00, bit de sistema en 0, tipo de puerta en 1 (32 bits) y el valor 110 en los bits 8-10. Por el momento, las rutinas de atención de excepciones estan hechas de manera tal que solo muestren por pantalla el error incurrido.

b) Cargamos la IDT agregando en `kernel.asm` un call a `idt_inicializar` y usando la instrucción `lidt` y la estructura `idt_descriptor` ya definida.

4. Ejercicio 3: Paginación Basica

a) Creamos la rutina `inic_video` en `screen.c`, esta sera la encargada de pintar la pantalla, limpiar el buffer de video e inicializar los valores en pantalla: ninguna tarea activa, puntajes nulos, mapa sin explorar.

b) Se escribe la rutina `mmu_inicializar_dir_kernel` que como su nombre indica, inicializa el directorio y las tablas de páginas del kernel. Como es necesario mapear desde `0x00000000` a `0x003FFFFF` tendremos que utilizar una sola tabla de páginas. Esta se conecta con la primer entrada del directorio, la cual se encontrará presente, mientras que el resto de las entradas no lo estarán. El directorio de páginas se inicializa en la dirección `0x27000` mientras que la tabla se inicializa en `0x28000`. El mapeo lo realizamos con *identity mapping* dentro del rango establecido.

c) Una vez inicializados el directorio y las tablas de páginas, se procede a activar la paginación en `kernel.asm`. Para esto se coloca en el registro `cr3` la dirección base del directorio de páginas (`0x27000`), esto nos asegura que los bits `PCD` y `PWT` del registro están limpios, y finalmente se coloca en 1 el bit m `PG` del registro `cr0`.

```
...
; Inicializar el manejador de memoria
call mmu_iniciar
;Inicializar el directorio de paginas
mov eax, dir_kernel_addr ; 0x27000
;Cargar directorio de paginas
mov cr3, eax
```

```
;Habilitar paginacion

mov eax, cr0
or eax, 0x80000000
mov cr0, eax
...
```

d) Creamos la funcion `imprime_nombre_grupo` en `screen.c` que calcula el tamaño del string Çircus / Family "(función auxiliar `long_string`) y en base a esto determina la posición que debe tener en la pantalla. El resultado puede ser observado en la figura 2.

Figura 2: Nombre de Grupo

5. Ejercicio 4: Memory Management Unit

a) La rutina `mmu_inicializar` en `mmu.c` inicializa en 0 un contador de páginas ocupadas páginas ocupadas, que se irá incrementando en la medida en que se utilicen las páginas, y llama a la función `mmu_inicializar_dir_kernel`, previamente creada.
Contamos también con la función `obtener_pagina_libre` que se encarga de incrementar el contador de páginas ocupadas y retornar el valor de una página libre reservada (estas páginas serán obtenidas del area libre del kernel). Para tratar de evitar problemas de solapamiento, resolvimos incrementar de a 2 el contador de páginas ocupadas, lo que terminaría en la reserva de 2 páginas. TAL VEZ HAY QUE CAMBIAR ESTO.

b) Se escribe la rutina `mmu_inic_dir_pirata` para inicializar un directorio de páginas y tablas de páginas para una tarea. Esta función solo se encargará de inicializar el directorio de páginas y las tablas para un tarea, en lugar de tener que también copiar el código de la misma, responsabilidades que dejamos para otras funciones. Su funcionamiento resulta muy similar al de inicializar el directorio y las tablas para el kernel. Difiere en el hecho de que se debe pedir una página libre para el directorio y las tablas de páginas, devolviendo la dirección de la página pedida para el directorio.

Para el trabajo de copiar el código del pirata y mapear las páginas correspondientes, se crean las funciones `copiar_código` y `mapear_alrededores` en `mmu.c`. La primera recibe como parámetros un cr3 (de la tarea cuyo código se quiere copiar), dos direcciones virtuales (una destino y otra origen) y una posicion pasada como 2 unsigned ints x e y (que serán pasadas como parámetros al código de la tarea). Esta función transforma las direcciones virtuales y con los otros parámetros antes mencionados llama a `copiar_fisico` la cual recupera el cr3Actual mediante la función `rcr3`, mapea ambas direcciones fisicas (DST y SRC) a las direcciones `0x500000` y `0x501000` (direcciones

que no tienen otro uso fuera de este) y tratandolas como punteros a int, se procede a copiar el código desde SRC a DST de a 4 bytes a la vez. Por último, `copiar_fisico` pasa los parámetros x e y a los últimos 8 bytes del destino y desmapea las 2 páginas mapeadas previamente. Para finalizar, mapeamos a la direccion `0x400000` la dirección física de destino. La segunda función, `mapear_alrededores`, toma un cr3 sobre el cual realizar los mapeos y una dirección virtual Destino. Se pasa luego a transformar la dirección virtual a física y hacer los mapeos, teniendo en cuenta los cálculos correspondientes, para que se mapeen las 9 casillas ocupadas mantiendo una correspondencia entre direccion virtual y física.

c) Se crean las rutinas encargadas de mapear y desmapear páginas de memoria.

La rutina `mmu_mapear_pagina` toma el valor de `cr3` y la direccióon virtual, sobre la cual calcula los índices correspondientes a directorio y tablas de páginas. Luego accede a la entrada del directorio y procede de la siguiente manera: si esta entrada se encuentra como *presente*, se accede a la entrada correspondiente de tabla y se realiza el mapeo con la dirección física indicada; en caso contrario, primero se completa dicha entrada pidiendo una página libre y luego se hace el mapeo propiamente dicho.

La rutina `mmu_unmapear_pagina` toma el valor actual de `cr3` y la dirección virtual, de la cual calcula los índices correspondientes a directorio y tablas de páginas. Luego accede a la entrada del directorio y, si estaba como *presente*, busca la entrada en la tabla de páginas y la pone como *no presente*. En el caso de que ya se encontrar como *no presente* no hace ningún cambio.

6. Ejercicio 5: Interrupciones De Teclado/Reloj

a) Usaremos la entrada número 32 de la IDT para la interrupción de reloj, la número 33 para la interrupción de teclado y la número 70 para la interrupción de software `0x66`. Las tres entradas se cargan como *interrupt gates*, y con los mismos atributos indicados en el apartado 2, con la diferencia de que a la 70 se le asigna el nivel de privilegio 3.

b) Se escribe una base para la rutina de atención. En esta instancia, el único objetivo de la rutina es llamar a la función `screen_actualizar_reloj_global`, la cual se encarga de mostrar la animación de reloj por cada tick de reloj.

c) Definimos la base de la rutina de atención de teclado de manera que al presionar cualquier tecla, muestre un mensaje en la esquina superior derecha de la pantalla (llamando a `imprime_tecla` definida en `screen.h`). Esto se realiza para corroborar el buen funcionamiento de las interrupciones, en el futuro, alteraremos este aspecto. Luego de esto, la rutina llama a la función `game_atender_teclado`, pasándole como parámetro el código de la tecla presionada. Esta se encarga de realizar la acción que corresponda dependiendo de la tecla indicada (lanzar explorador o activar debug).

d) Se escribe la rutina de atención de la interrupción numero `0x46` para que mueva un `0x42` al registro EAX.

7. Ejercicio 6: TSS

a) Definimos las entradas 13 (tarea inicial), 14 (tarea idle), 15-22 (tareas jugador 1), 23-30 (tareas jugador 2) de la GDT A estas entradas se les asigna tipo 9 (Execute-Only,accessed), base 0, presente 1 y DPL 0. El límite se establece en `0x67` para que sea mayor al tamaño de una TSS. La Figura **??** muestra el estado de la GDT luego de agregar estas entradas.

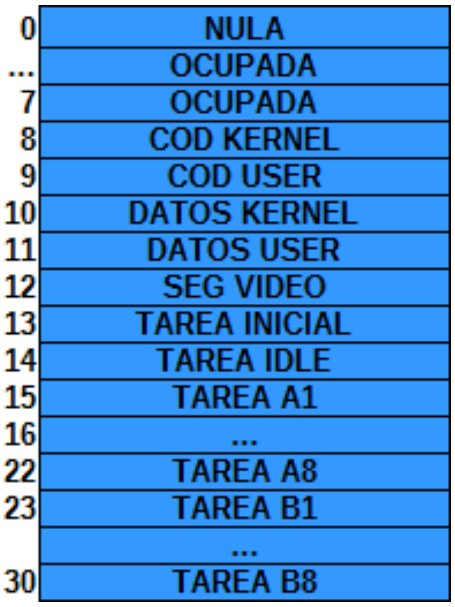

Figura 3: Estado final de la GDT

b) Para completar la TSS de la tarea Idle hacemos uso de la función `tss_inicializar_tarea_idle`, la cual se encarga de asignar los segmentos de datos del kernel a los campos GS, FS, DS, SS, ES y poner el segmento de código de kernel en el campo CS. A su vez, completa ESP y EBP con la dirección del stack del kernel y se ocupa de que comparta el cr3 con el kernel. Por último, el EIP será `0x16000`, el campo EFLAGS se completa con `0x202` (para habilitar las interrupciones), el IOMAP con `0xFFFF` y el resto se deja en 0.

c) La función `tss_inicializar_tareas_piratas` toma como parámetro un puntero a tss y se ocupa de llenar sus campos. Los campos ES, SS, DS, FS y GS se completan con el segmento de datos de usuario, y el campo CS se completa con el segmento de código de usuario. ESP y EBP quedan seteados en `0x401000-12` y `0x401000-4` respectivamente, el campo EFLAGS en `0x202`, el CR3 en 0 (se le asignará un cr3 al momento de lanzar el pirata correspondiente utilizando la función `mmu_inicializar_dir_tarea`), se pedirá una página nueva para el ESP0, SS0 se completa con segmento de datos kernel, IOMAP con `0xffff` y el resto con 0.

d) En la rutina `tss_inicializar` pedimos una página libre para la tss de la tarea inical y asignamos la dirección obtenida en el campo *base* de la entrada correspondiente a esta tarea en la GDT.

e) En la rutina `tss_inicializar` completamos la entrada de la GDT perteneciente a la tarea Idle, completando el campo *base* con la dirección correspondiente a la misma.

f) Para pasar a la tarea IDLE primero se carga la tarea inicial en `kernel.asm`, haciendo uso de la instruccion `ltr` y de la dirección del segmento de la tss inicial. Una vez cargada la tarea inicial, se ejecuta un `jmp` a la etiqueta seg_tss_idle, que corresponde al segmento de la tss Idle.

```
define selector_Inicial 0x0068 ;0000000001101011
define selector_Idle 0x070 ;0000 0000 0111 0011

...

; Cargar tarea inicial
mov ax, selector_Inicial
ltr ax

; Saltar a la primera tarea: Idle
jmp selector_Idle:0
```

```
...
```

g) Se modifica el comportamiento de la rutina de atención de la interrupción `0x46` de la siguiente manera. Se pasan los 2 parámetros a `game_syscall_manejar` y se la ejecuta. Luego de haber terminado `game_syscall_manejar` se encargará de saltar a la tarea idle.

La función `game_syscall_manejar` determina en base al primer parámetro tomado si se quiere ejecutar un syscall mover, cavar o posición y llama a `game_syscall_pirata_mover`, `game_syscall_cavar` o `game_syscall_pirata_posicion` según corresponda. Analizaremos por separado los 3 casos:

- Caso mover: Según el jugador y el índice de la tarea que se encuentren jugando en el momento se busca el pirata que corresponde en el arreglo de piratas del jugador (piratasA o piratasB) así como también el arreglo de posiciones visitadas del jugador (visitadasA o visitadasB). Una vez obtenidos estos, se busca la posición actual del pirata y se cheque si el movimiento que desea realizar es válido o no (en caso de que no lo sea se llama a la función `game_matar_pirata_interrupt`, en caso de ser válido se calcula la posición destino y la dirección virtual asociada a esa posición, se busca el cr3 de la tarea en su tss y se actualiza la posición del pirata. En el caso de que el pirata fuera un minero se llama a `revisar_mapeadas` que como indica su nombre, revisa si la posición virtual destino ya habia sido explorada, en caso de que no lo fuera, se mata a la tarea. Pasado este punto se llama a `copiar_codigo` con el cr3, la posición virtual destino, la posición virtual asociada a la posición anterior y las componentes x e y de la posición destino. En caso de que estuvieramos hablando de un explorador, este mapea las nuevas direcciones exploradas a todas las tareas del jugador, actualiza el arreglo de visitadas y el indice de la última posición válida del arreglo y pasa a buscar botines. Para buscar los botines, se revisa si hay algun elemento en el arreglo de botines que entre en el rango recientemente explorado y contenga monedas, de ser así, se lo pinta en el mapa y se llama a `game_jugador_lanzar_pirata` que lanzara una tarea pirata para el jugador, de tipo minero, y con la posición del botín descubierto. Para finalizar, se llama a `screen_pintar_pirata` con un puntero al jugador y al pirata, y se lo pinta con los colores correspondientes en el mapa.
- Caso cavar: Según el jugador y el índice de la tarea que se encuentran jugando en el momento se busca el pirata que corresponda y se obtiene su posición. Con esta posición se recorre el arreglo de botines buscando alguno que encaje en la posición del minero. En caso de que el botin ya no tenga mas monedas, se mata a la tarea. Caso contrario, se aumenta el puntaje del jugador en 1, se actualiza el puntaje en pantalla y se decrementa la cantidad de monedas en el botín en 1.
- Caso posición: Según el jugador que se encuentra jugando, y el parámetro de la función, se va al arreglo de piratas correspondiente al jugador y segun el parámetro se busca el pirata cuyo índice corresponda (en caso de ser 0-8) o el pirata actual (en caso de ser -1). Una vez obtenida la posición del pirata, se crea una variable resultado, se le suma la componente y, se shiftea 8 lugares y se le suma la componente x. Se termina retornando dicha variable.

h) Para este apartado hicimos algo más que lo pedido por el enunciado. Probamos lanzar una tarea explorador a través los siguientes pasos: llamar a `inic_game` (función que se encarga de inicializar las variables globales del juego como los jugadores con sus respectivos puertos, 0 tareas activas, etc y todas las tareas piratas correspondientes a cada jugador como muertas y con sus respectivos indices, entre otras cosas), luego, mediante `tss_inicializar` realizamos lo concerniente a la tss de las tareas (dejando lugar para un cr3 que se creara luego), y para finalizar ya dentro del juego presionamos la tecla RSHIFT (cual shift se presiona resulta indiferente para los efectos de la prueba). Presionando RSHIFT se llama a `game_lanzar_pirata` que se encarga de inicializar el mapa de memoria de la tarea, copiar su codigo en el mapa y completar su campo posicion, además de mostrarla en el mapa de juego. Por último se procede a correr esa tarea previamente lanzada.

8. Ejercicio 7: Scheduler

a) Se inicializan las variables globales proximaA y proximaB (que se usaran para indicar el indice de la proxima tarea del jugador) en 0, jugadorJugando en 0, actual (que sera el indice de la tarea que esta corriendo, será 0-8), y gdt_tss_actual en 13 (la entrada 14 corresponde a la idle y de ahi en adelante a las tareas del jugador 1 y 2).

b) En este apartado nos tomamos ciertas libertas con respecto a la organización de las funciones del scheduler, a continuación detallaremos el uso de cada una. La función `sched_proxima_a_ejecutar` es renombrada como `sched_proximo_indice`. Esta llama a `obtener_proxima_viva` la cual hace los chequeos de jugadorJugando (si el jugadorJugando es 0, osea el jugador 1, buscará una tarea viva del otro jugador, y viceversa), en caso de encontrar una tarea viva del otro jugador, actualiza jugadorJugando a dicho jugador y devuelve el indice de la tarea, en caso de no encontrar una tarea viva devuelve `0xff`. El resultado de `obtener_proxima_viva` se lo compara con `0xff` y en ese caso se llama a `pasar_a_idle` y se retorna `0xff`. En caso contrario, se llama a `cambiar_tarea` con la proxima tarea viva, se actualiza el clock de la tarea en el mapa y se devuelve el indice en la GDT de la tss de la tarea a la que se pasa.

   El funcionamiento `pasar_a_idle` pasa por revisar si actual es `0xff` y en ese retorna `0xff` terminando con la ejecución (dado que la idle no debe ser capaz de hacer un cambio de tarea a si misma). En caso contrario, se llama a `cambiar_tarea` con `0xff` y se retorna el indice en la GDT de la TSS de la tarea IDLE.

   El cambio de tarea realizado por `cambiar_tarea` consta de comprobar si el parametro tomado es igual a `0xff` y en ese caso se modifica actual con este valor y gdt_tss_actual pasa a ser 1 (luego con un calculo de $gdt\_tss\_actual + 13 << 3$ en `sched_proximo_indice` se obtendría el indice de la tss en la GDT). En caso contrario, actual pasa a ser igual al parámetro tomado por la función se resetean proximaA y proximaB según corresponda, en caso de superar el número de tareas (en caso de ser mayor a 8 vuelve a 0). Por último se obtiene el índice de la tss de la tarea actual en la GDT y se lo asigna a gdt_tss_actual.

d) Se modifica la rutina de atención de la interrupción `0x46` para que luego de llamar a `game_syscall_manejar` desaloje a la tarea que la llamo mediante los siguientes pasos: se llama a `pasar_a_idle` y un jmp far al selector que devuelve esta, con el offset 0.

e) La rutina de atención se modifica para que ejecute los siguientes pasos, permitiendo el intercambio de tareas por cada ciclo de clock:

   - Se guarda en un registro el resultado obtenido en `sched_proximo_indice`
   - Se lo compara con `0xff`, valor que usamos para identificar a la tarea idle. En este caso, indica que no quedan piratas vivos para ejecutar. De ser iguales no se produce salto alguno.
   - Se lo compara con el valor previo del selector, si son iguales quiere decir que es la única tarea activa. Como una tarea no debería ser capaz de saltar a si misma, no se produce salto alguno y continúa corriendo la tarea.
   - Si ninguno de los 2 casos anteriores se cumple, se procede a ejecutar un JMP FAR al resultado de realizar `sched_proximo_indice`, al offset correspondiente en la GDT a la tss de la tarea destino.

f) Para manejar el hecho de desalojar una tarea cuando esta produce una excepción implementamos, en cada rutina de atención de excepciones, un llamado a la función `game_matar_pirata_interrupt`. Esta función, por un lado llama a la función `game_matar_pirata` la cuál se encarga de hacer lo relacionado con mostrar el clock del pirata como muerto y setear en 0 el atributo vivo del pirata, de manera que no se puedan producir saltos a esta tarea. Por otro lado, revisa si había algun minero pendiente en el arreglo de mineros pendientes del jugador en cuestión, de ser así lo lanza con la posición del botín previamente almacenada en dicho arreglo y actualiza sus valores.

Figura 4: Ventana debug

g) El modo debug permite capturar y mostrar por pantalla las excepciones causadas por la operación de los piratas. El mismo se activa presionando la tecla Y del teclado. Una vez activo, el juego continúa de manera normal, esperando a que se produzca la próxima excepción por parte de una tarea cualquiera. Cuando esto suceda, se mostrará un cuadro con la información correspondiente a la tarea que experimentó la falla como se muestra en la siguiente figura.

   Al presionarse nuevamente la tecla Y, el cuadro de debug desaparece, pudiendo continuarse con el juego, siempre a la espera de la próxima excepción. En la pantalla se muestra la representación hexadecimal del estado de los registros eax, ebx, ecx, edx, esi, edi, ebp, esp, eip, cs, ds, es, fs, gs, ss, cr0, cr2, cr3 y cr4. Además, mostramos la representación del registo eflags y de los 4 valores superiores de la pila, también en hexadecimal. Para regular este modo, tomamos la decisión de crear una variable global debug que posee 3 estados distintos: (0) para debug no activado, (1) para debug activado, (2) para modo mostrando. Al iniciar el juego esta variable se setea en 0, el pasaje a 1 se regula por la atención de teclado, al apretar la tecla correspondiente, y el pasaje al modo 2 se realiza al capturar la primer expepción que ocurra luego de pasar a modo 1. En el caso de estar mostrando y presionar la tecla Y, se restaurará la pantalla y se continuará con el juego. Para restaurar la pantalla, almacenamos previamente el estado de la pantalla en el sector ocupado por la ventana de debug en un buffer. Este buffer se encuentra en el área de memoria libre a la que denominamos `VideoCache`. Modificamos las rutinas de atención de las excepciones, agregando las lineas necesarias para, si el modo debug se encontraba activado, llamar a la función `screen_muestra_error` luego de matar a la tarea que produjo el error. Esta función se encarga de copiar al buffer de video el sector de la pantalla que se verá modificado y muestra la ventana con la información exigida y setea la variable `debug` en 2 para indicar que se encuentra mostrando. A la función `screen_muestra_error` previamente mencionada, se le pasan por parámetro los valores de los registros, y otros valores a mostrar. Ya dentro de la función, y despues de copiar la pantalla, obtenemos los valores de los registros (cr0, cr2, cr3, cr4). Mientras el cuadro se encuentra en pantalla, la función `obtener_proxima_viva` del scheduler devuelve siempre el valor `0xFF`, indicando que la ejecución debe quedarse en la tarea *Idle*. Dicho comportamiento se conserva hasta que se presiona la tecla Y, con la cual se reanuda el juego. Además, vale aclarar que desde la rutina de atención del teclado, se realizan las comparaciones necesarias sobre la variable debug para que no suceda evento alguno (no se podrán lanzar nuevas tareas), salvo que se presione la tecla Y reanudando el juego.

9. Ejercicio 8: Finalizar el Juego

El juego finalizará cuando se cumplan alguna de las condiciones listadas a continuación:

- Se agotan todos los botines del juego.
- Pasan 999 ciclos de clock sin que suceda ninguna acción por parte de algún jugador. Esto comprende tener tareas activas.

Para chequear estas condiciones se incorpora la linea `game_ver_si_termina` a la rutina de atención a la interrupción de reloj. Es necesario aclarar que esta rutina, de no producirse salto alguno, incrementa en 1 un contador denominado `tiempo_sin_juego`, siendo reseteado al lanzar alguna tarea. En caso de finalizar el juego, se llamará a `screen_stop_game_show_winner`, que como su nombre indica, mostrará por pantalla una ventana del color del jugador victorioso y su puntaje.