DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Programación de Sistemas, Lo de Charly Bondiola

Organización del Computador II
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Federico De Rocco	403/13	fede.183@hotmail.com
Fernando Abelini	544/09	ferabelini@outlook.com
Manuel Casanova	355/05	casanova_manuel@yahoo.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

Índice

1. Introducción

En el siguiente informe se mostrara y explicara la forma en que se lograron resolver los ejercicios donde se utilizan los conceptos de System Programming. El sistema busca poder correr 16 tareas al mismo tiempo a nivel de usuario. Para esto se implementara un juego sencillo de dos jugadores que correrá usando este sistema. Cada jugador podrá enviar un zombie para atacar a su oponente. Por zombie nos referimos a tareas del sistema que pueden desplazarse a través de la memoria buscando a otras tareas para devorar sus cerebros. Ambos jugadores tienen un máximo de 8 zombies en juego de tres clases, Guerrero, Mago y Clérigo.

2. Ejercicio 1

2.1. Tabla de Descriptores Globales

También llamada GDT es una estructura que nos permitirá gestionar y direccionar segmentos de memoria(Tanto de datos como de programas). En el caso puntual del tp, está contara con 4 segmentos, tendremos dos de ellos para el código y los otros para datos, cuyos niveles serán 0 y 3 respectivamente. Estos segmentos direccionan los primeros 623MB de memoria. Una restricción impuesta para este trabajo práctico nos dice que las primeras 7 posiciones no pueden utilizarse y a los ojos del enunciado están ocupadas. Entonces el primer índice usado para declarar estos segmentos es el número 8, exceptuando el descriptor nulo que está en la posición 0. Este último será usado para cargar la GDT.

En la GDT se construyen 6 segmentos. El primero es la dirección que se encuentra vacía. Los de código y datos poseen en como límite 623MB-1(osea los primeros 623Mb direccionados comenzando desde el 0). De estos la única diferencia es el type(Este lógicamente las distingue entre código y datos). Entre estos el primer, tanto en código como en datos, tiene el dpl marcado para que solo pueda usarlo el kernel. Las otras pueden ser usadas por los procesos o tareas. El último segmento corresponde a la pantalla. La misma tiene un tamaño de 80x50 píxeles. Por esto, su límite debe ser de 80*50*2 -1 y su base está marcada en la dirección 0xB8000. También lo pusimos como datos, usando type. Para poder verlo en bytes usamos que la granularidad es 0(bit g del segmento).

2.2. Pila

Se debe setear la pila en la posición 0x27000(Pasado por enunciado) para pasar a modo protegido. Esto se hace asignandole este valor a nuestros registros base y tope de pila(ebp y esp).

2.3. Pintar Pantalla

El enunciado nos pidió hacer una rutina que se encargue de dibujar la pantalla con color verde. Para resolver esto se construyó una macro en el archivo pantalla.mac. Dicha macro escribe es cada píxel de la pantalla el valor 0x2020. El primer byte de valor 20 tiene asociado el color, el cual es verde, el segundo tiene asociado el valor del carácter espacio. El primer ciclo recorre toda la pantalla. El segundo recorre solo la primer columna de esta. En el mismo se asigna el valor 0x4020. Este representa nuevamente al carácter espacio pero de color rojo. Con azul es equivalente, solo que tomamos la última columna y la pintamos de azul(0x1020). Algo importante de señalar es como se accede a la pantalla. Esto se realiza usando el selector de segmento fs.

2.4. Pasar a modo protegido

Para poder realizar esta acción tendremos que completar la GDT como se vio en la sección anterior y después se deben realizar los siguientes pasos:

1. Deshabilitar interrupciones(Usando CLI).
2. Cargar el registro GDTR con la dirección base de la GDT.

3. Setear el bit PE del registro CR0.
4. Hacer un FAR JUMP a la siguiente instrucción.
5. Cargar los registros de segmento(DS, ES, GS, FS y SS).

En esencia solo se debería realizar el paso 3 pero hay que deshabilitar las interrupciones para que no caiga una justo en medio del proceso. También se deberá hacer el paso 2 para poder estar seguro que tenemos la gdt. Usando el FAR JUMP ponemos el límite de lo que se ejecuta en modo real y después de este estamos ejecutando en modo protegido. Por último establecemos los selectores de segmentos enumerados en el punto 5.

Hay algunas cosas que debemos hacer antes de comenzar el pasaje a modo protegido. Esto es habilitar A20 la cual se define como una linea de control para el controlador de memoria, esta nos habilita el acceso a direcciones superiores a los 2 a la 20 bits. Este pin se debe habilitar para poder direccionar sobre el MB de memoria.

3. Ejercicio 2

3.1. IDT

Para completar la estructura utilizamos la macro IDT_ENTRY. La cual recibe un número que corresponde a la interrupción en la tabla de interrupciones. La función idt_inicializar se encarga de usar esta macro para cada una de las excepciones de la tabla que nos interese poner. Estas son únicamente las que están contempladas por INTEL. Las demás están reservadas. Estas nos quedan de la siguiente forma:
$IDT\_ENTRY(0)$
$IDT\_ENTRY(2)$
$IDT\_ENTRY(3)$
$IDT\_ENTRY(4)$
$IDT\_ENTRY(5)$
$IDT\_ENTRY(6)$
$IDT\_ENTRY(7)$
$IDT\_ENTRY(8)$
$IDT\_ENTRY(9)$
$IDT\_ENTRY(10)$
$IDT\_ENTRY(11)$
$IDT\_ENTRY(12)$
$IDT\_ENTRY(13)$
$IDT\_ENTRY(14)$
$IDT\_ENTRY(16)$
$IDT\_ENTRY(17)$
$IDT\_ENTRY(18)$
$IDT\_ENTRY(19)$

Para poder atender las excepciones se fabricó un conjunto de mensajes para cada una de las ellas. Dichos mensajes se mostraran en pantalla usando la macro ISR la cual consiste en usar el número de excepción y llamar a la función imprimir_texto_mp pasándole como dato el mensaje adecuado para esta excepción. Además de la parte superior izquierda de la pantalla, el color del texto y fondo.

También debemos completar la isr.h con cada una de las funciones pertinentes de las excepciones anteriores. Estas serán posteriormente definidas en isr.asm usando la macro anteriormente mencionada.

4. Ejercicio 3

4.1. Pantalla

En esencia se hace lo mismo que en el Pintar Pantalla del ejercicio 1. Lo único que se agrega es la parte negra de la parte baja y los cuadros azul y rojo de cada jugador que indican el puntaje. También imprimimos los números que van por encima de los relojes de los zombies.

4.2. Inicializar directorio de páginas y tabla de páginas del kernel y activar paginación

El directorio de páginas se inicia en la posición 0x27000 y la tabla de páginas en la dirección 0x28000. El valor de la base del directorio de páginas se define con el registro cr3, solo que teniendo en cuenta solo los primeros 20 bits(Tiene que ser múltiplo de 4kb). Todos los demás bits de atributos los pusimos en 0. Simplemente cargamos ese valor. PCD y PWT tienen que estar desactivados para poder activar paginación. Completamos el directorio de páginas con páginas en blanco(todos los valores nulos). Excepto la primera que corresponde a la de la tabla de páginas. La tabla de páginas fue completada con páginas con todos sus atributos en 0, excepto el de presente y el de lectura escritura que van en 1. En la base ponemos valores ascendentes del 0 al 1023. Con la intención de mapear los 4mb de memoria.

Para poder activar la paginación solo nos resta setear el bit PG del cr0. Esto se debe hacer solo si se realizaron los pasos anteriores, osea armar el directorio de páginas y tabla de páginas.

5. Ejercicio 4

5.1. Inicializar mmu

Para esta función utilizaremos la variable global indice_pagina_libre la cual, como el nombre lo indica, nos dirá la posición de una página libre en la tabla de páginas. Usaremos la función inicializar_mmu para cargar en esa variable la posición de la primer página libre. Esta se encuentra en 0x00100000.

5.2. Inicializar un directorio de páginas y tablas de páginas para una tarea

Usaremos las estructuras offset_pagina_zombie_A y offset_pagina_zombie_B las cuales nos servirán para saber que posición ocupa cada página de la tabla en la pantalla.

Para poder realizar lo pedido debemos tener como parámetros el jugador y la posición en pantalla. Comenzaremos guardando en nuestra variable cr3 la dirección de una nueva página, posteriormente la usaremos para mapear las páginas. Obtendremos esto usando la función obtener_pagina_libre la cual aumenta el índice de página libre y lo devuelve. Usando esta variable cr3 mapeamos las páginas con idetity mapping las direcciones de 0x00000000 a 0x003fffff.

Dependiendo de que jugador sea, mapearemos las direcciones desde 0x08000000 a 0x08008000 usando la función mmu_mapear_pagina y las estructuras offset_pagina_zombie_A y offset_pagina_zombie_B. Para cada caso debemos conseguirnos la posición inicial de mapa.
Para el jugador 1: $pos\_inicial\_mapa = ((posicionY * 78 + 1) * 0x1000) + base\_mapa$
Para el jugador 2: $pos\_inicial\_mapa = ((posicionY * 78 + 77) * 0x1000) + base\_mapa$

Siendo posicionY la pasada como parámetro y base_mapa el valor 0x00400000 que hace referencia a la primer posición libre después de la memoria del kernel. Utilizaremos esta posición inicial para poder obtener la dirección física. Sumaremos este valor con cada una de las ocho posiciones de las estructuras offset para obtener las direcciones físicas a mapear en la pantalla. Y pondremos en cada una de ellas el valor de atributos en 1, para setear el bit de Usuario Supervisor, y como dirección virtual tomaremos las posiciones 0x08000000 a 0x08008000. También usaremos el cr3 obtenido anteriormente.

5.3. Mapear página

Haremos esto tomando una dirección física, una virtual, un cr3 y los atributos. Tomamos la dirección y la descomponemos en sus partes y solo usaremos el índice del directorio y de la tabla. Nos conseguimos la base del directorio de páginas con el cr3. Después usaremos la parte alta de la dirección virtual para obtener el la correspondiente PDE. Tenemos que asegurarnos que la página este presente. Si no lo está, seteamos el bit p, ponemos en base una página libre y le asignamos los atributos pasados. Después obtendremos la base de la tabla de páginas de ese directorio y usaremos la parte de table de virtual para obtener la PTE buscada. Seteamos el bit de presente, le asignamos en base la dirección física y le ponemos los atributos. Concluiremos llamando a la función tlbflush para invalidar la cache de traducción de direcciones. Como aclaración tenemos que mencionar que atributos solo contiene información sobre el us(Usuario/Supervisor).

5.4. Unmapear página

Para esta función haremos lo mismo que en la anterior. Osea buscar la PDE y, posteriormente, la PTE correspondiente usando el cr3 y la dirección virtual. Después tenemos que poner el bit de presente de esa tabla en 0. Si esta página resulta ser la última página presente en la tabla de páginas, también pondremos en 0 el bit p del directorio de páginas correspondiente a esta tabla.

6. Ejercicio 5

En este punto, agregamos las entradas 32, 33 y 102(0x66) en la IDT. Estas se utilizan para clock, teclado y software respectivamente.

En los 3 casos se realizan los siguientes pasos:

1. Deshabilitar las interrupciones.
2. Preservar los registros.
3. Comunicar al PIC que ya se atendió la interrupción(salvo la 102 que no se comunica con el pic).
4. Realizar la tarea correspondiente a la interrupción.
5. Restaurar los registros.
6. Habilitar las interrupciones.
7. Retornar de la interrupción.

6.1. Reloj

En cada interrupción de clock se ejecuta la función proximo_reloj. Dicha función es la que se encarga de imprimir el reloj en la esquina inferior derecha de la pantalla de juego.

6.2. Teclado

Cada vez que se pulsa una tecla de las habilitadas para el juego, el pic nos informa que se produjo una interrupción. Mediante el puerto 0x60 obtenemos el scan code de la tecla que fue presionada. En el archivo pic.c creamos la función que convierte un scan code(de los 10 habilitados para el juego), en un carácter para luego ser impreso en pantalla. En caso de ser una tecla que no es del juego, se ignora.

La macro imprimir_texto_mp recibe la dirección de memoria del mensaje. Con lo cual en el código de la atención de interrupción del teclado, pusheamos el carácter en la pila y utilizamos la dirección de ESP como parámetro.

7. Ejercicio 6

7.1. GDT de tareas

Se pide describir dos gdt para tareas. Una para la tarea inicial y otra para la tarea idle. En el caso de la tarea inicial:
$limit = No\ importa$
$base = No\ importa$
$type = 0x09$
$s = 0x00$
$dpl = 0x00$
$p = 0x01$
$avl = 0x00$
$l = 0x00$
$db = 0x00$
$g = 0x01$

No nos importa que tenga en base o límite porque no lo vamos a usar. En el caso de base la vamos a inicializar desde tss.c. El tipo tiene que ser 9 ya que debe poderse ejecutar y además acceder. El s debe estar en 0 ya que esto pertenece al sistema y no a código o datos. El dpl debe estar en 0 porque, como ya dijimos, esto es usado por el sistema entonces debe tener nivel de privilegio kernel. El uso de esta tarea es basicamente servir para respaldar el contexto de una tarea cuando cambie a otra. Obviamente en ambos casos debe tener el bit presente seteado. Lo demás queda como los casos de datos y código anteriores, granularidad en 1, avl en 0, etc.

Para la tarea idle, la gdt quedaría asi:
$limit = 0x000067$
$base = 0x00000000$
$type = 0x09$
$s = 0x00$
$dpl = 0x00$
$p = 0x01$
$avl = 0x00$
$l = 0x00$
$db = 0x00$
$g = 0x01$

Todos los atributos son iguales a los de la tarea inicial. La única diferencia es que el límite ahora si nos va a importar, ya que esta tarea hace algo mas que solo servir para inicializar las tareas. Como en el anterior el valor de la base va a ser pasado posteriormente.

7.2. Inicializar la TSS

El código en C nos provee de dos variables que usamos para definir la tss. Estas son tss_inicial y tss_idle. Lo que debemos hacer con estas es completar los espacios base de las entradas de las gdt para las tareas del mismo nombre. Simplemente tomamos las direcciones de memoria de las variables y las asignamos como se ve a continuación.
$unsigned\ int\ base = \&tss\_inicial$
$base = base << 16$
$base = base >> 16$
$gdt[TAREA\_INICIAL].base\_0\_15 = base$
$base = \&tss\_inicial << 8$
$base = base >> 24$
$gdt[TAREA\_INICIAL].base\_23\_16 = base$
$base = (unsignedint)\&tss\_inicial >> 24$

$gdt[TAREA\_INICIAL].base\_31\_24 = base$

La esencia es tomar la dirección de memoria de la tss correspondiente y ponerla en base de la gdt. Lo mismo se hace para la tarea idle. Con la diferencia que en el caso de esta última tendremos que completar los atributos que en el caso de la inicial no nos interesaba como quedaban. Pasaremos a mostrar que asignamos en los campos de la tss_idle, los que no mostramos los dejamos en 0 porque no nos importaban.
$tss\_idle.esp0 = 0x17000$
$tss\_idle.ss0 = 0x50$
$tss\_idle.cr3 = 0x27000$
$tss\_idle.eip = 0x00016000$
$tss\_idle.eflags = 0x00000202$
$tss\_idle.esp = 0x27000$
$tss\_idle.ebp = 0x27000$
$tss\_idle.es = 0x50$
$tss\_idle.cs = 0x40$
$tss\_idle.ss = 0x50$
$tss\_idle.ds = 0x50$
$tss\_idle.fs = 0x50$
$tss\_idle.gs = 0x50$
$tss\_idle.iomap = 0xFFFF$

El cr3, es, cs, ss, ds, fs, gs, esp y ebp son los mismos que el kernel. La eip y iomap están pasadas por enunciado. El eflags tiene solamente seteado lo necesario para tener las interrupciones habilitadas.

7.3. Completar TSS libre

Pasando como parámetros el índice en la gdt, un jugador, la posición del jugador y una tss libre. Lo que haremos es similar a lo que hicimos en los enunciados anteriores. Primero obtenemos la base usando la tss libre y la asignamos a la gdt señalada por ese índice. Después completaremos los atributos de la tss.
$t->esp0 = obtener\_pagina\_libre() + 0x1000$
$t->ss0 = 0x40$
$t->cr3 = mmu\_inicializar\_dir\_zombie(posicionY, jugador)$
$t->eip = 0x00800000$
$t->eflags = 0x00000202$
$t->esp = 0x00801000$
$t->ebp = 0x00801000$
$t->es = 0x5B$
$t->cs = 0x4B$
$t->ss = 0x5B$
$t->ds = 0x5B$
$t->fs = 0x5B$
$t->gs = 0x5B$
$t->iomap = 0xFFFF$

El ss0 es el stack de nivel 0 y lógicamente debe ser el mismo que el del kernel. En esp0, que es el segmento para entrar en la pila ss0, la obtenemos como una nueva página. La eip debe ser entre 0x00800000 porque son de las tareas zombie. Los registros esp y ebp tienen que usar la parte del final de su página de código. El eflags está igual que el anterior, con las interrupciones habilitadas. Los selectores de código y datos tienen por valor las posiciones de código y datos 2 de la gdt. El cr3 es el que obtenemos para asignarle una tarea zombie usando mmu_inicializar_dir_zombie. Para esto usamos el jugador y la posición del mismo, pasadas como parámetros.

8. Ejercicio 7

8.1. Estructuras

Las estructuras de datos que necesitaremos para el scheduler serán las siguientes:

1. Un entero de nombre sched_jugador que será 0 si el jugador actual es el primero, 1 si es el segundo, -1 si se está corriendo la tarea idle.
2. Un entero sched_indice_tarea que indique el índice del zombie vivo, actualmente corriendo.
3. Un entero sched_indice_tarea_A que nos dirá el índice del último zombie del primer jugador.
4. Un entero sched_indice_tarea_B ídem anterior para el segundo jugado
5. Un entero sched_indice_actual que nos dice que índice de la gdt que se está ejecutando.

Para el juego usamos una estructura jugador que tiene los siguientes datos:

1. Un entero tipo_zombie que nos dirá cual es el tipo de zombie que está marcando en el lanzador. Si vale 0 es guerrero, 1 es mago y 2 es clérigo.
2. Un entero de nombre pos que nos dirá la posición donde se encuentra el zombie en el lanzador. Como solo hay dos jugadores, al saber que jugador es ya sabemos la posición de la columna y esta variable nos indica la fila.
3. Un entero zombies_lanzados que nos indicará la cantidad de zombies que fueron lanzados por ese jugador. El valor nunca es superior a 20 por como está implementado game_lanzar_zombie.
4. Un arreglo de zombie_descriptor nombrado como zombies. El arreglo tiene tamaño 8(Cantidad máxima de zombies de un jugador en juego). Esta estructura se describirá posteriormente.
5. Un entero colinicial que nos indicará cual es la columna donde se lanzará el zombie del jugador. Si es el primer jugador será 1, 76 para el segundo.
6. Un entero puntaje el cual tendrá la cantidad de veces que al jugador contrario le entró un zombie en su área(Anota un punto).

Agregamos 16 descriptores para la gdt que nos servirán para cada una de las tareas de los zombies. La razón por la cual son 16 es porque cada jugador puede tener una cantidad máxima de 8 tareas en juego.

Para el zombie usamos la estructura zombie_descriptor que tiene los siguientes datos:

1. Un carácter de nombre tipo. Que tiene como valor 'G' si el zombie es guerrero, 'M' si es mago y 'C' en el caso que sea clérigo.
2. Un entero de nombre vivo, que tendrá como valor 1 si el zombie está activo, 0 en caso contrario.
3. Un short de nombre gdt_index que, como el nombre índica, será el índice de la gdt de la tarea del zombie.
4. Un entero de nombre x, que será el índice de columna en el cual se encuentra el zombie en la pantalla.
5. Un entero de nombre y, ídem anterior para fila.
6. Un entero pos_reloj. Este dato se utiliza para imprimir los relojes en cada zombie y se actualiza por cada interrupción de reloj. Este tendrá de valor: 0 si el reloj está en -, 1 está en /, en 2 es | y en 3 está \.

8.2. Inicializar Scheduler

Debemos inicializar las estructuras del sched mencionadas anteriormente. Para esto les asignamos los siguientes valores:

1. $sched\_jugador = -1$
2. $sched\_indice\_tarea\_A = 0$
3. $sched\_indice\_tarea\_B = 0$
4. $sched\_indice\_tarea = 0$
5. $sched\_indice\_actual = (GDT\_IDX\_TAREA\_IDLE * 8)$ (Comenzamos con la tarea idle).

Como se dijo en la sección anterior, cuando sched_jugador está en -1 significa que se está corriendo la tarea idle. Por esto mismo ponemos sched_indice_actual en el índice de la gdt de la tarea idle.

8.3. Scheduler próximo índice

Esta función es la encargada de devolver el próximo índice a ser ejecutado. Devuelve un selector de segmento en caso de tener que saltar a una tarea, 0 si no hay que realizar un cambio de tarea. El selector devuelto se guarda en la variable global sched_indice_actual. el selector devuelto apunta siempre un tss_descriptor, y este puede pertenecer a un zombie o la tarea idle.

Esta función realiza los siguientes pasos:

1. Ordena a los jugadores, dependiendo de quien movió último. Esto lo determina utilizando la variable sched_jugador.
2. Busca dentro de los zombies del siguiente jugador el siguiente zombie vivo, si encuentra alguno, marca en las variables que va a ser el próximo zombie a correr. Para determinar cual es el siguiente empieza a buscar a partir del siguiente índice, dentro del vector de zombies, que corrió último para ese jugador. Para esto utiliza las variables sched_indice_tarea_A o sched_indice_tarea_B según corresponda.
3. Si no encuentra alguno vivo, realiza la misma búsqueda dentro del vector de zombies del otro jugador, y realiza las modificaciones correspondientes.
4. Si no encuentra algún zombie vivo, determina que la tarea que debe correr es la tarea idle, no modifica ninguna variable.
5. Por último, compara que el dato que determinó debe ser el siguiente. Para esto usa el valor de la tarea que se está ejecutando (sched_indice_actual), y si son iguales devuelve 0, indicando que no es necesario un cambio de tarea.

8.4. Salto de tareas

El salto de tareas se realiza en la interrupción de reloj. Modificamos esta interrupción agregando la función anterior y nos aseguramos de modificar los relojes de las tareas. Si la función sched_proximo_indice nos devuelve un valor distinto a 0 entonces se debe realizar el salto a la tarea que devuelve como resultado. En caso contrario no se realiza el salto y sale de la interrupción.

8.5. Interrupción 0x66

Debemos modificar esta interrupción para que implemente el servicio mover. Como sabemos la syscall nos pasará en eax un dato que nos dirá para donde debe desplazarse el zombie actualmente en juego. Utilizamos una función en C llamada zombie_mover para poder resolver este problema. Dicha función recibe el dato en eax y hace lo siguiente:

1. Primero que nada nos creamos cuatro variables: adelante, atras, izquierda y derecha. Que contendrán los valores en que se deberá desplazar el zombie en pantalla para cada caso, estas serán usadas para cambiar los valores de x ó y de los zombies. Tomamos en cuenta también los casos borde.

2. Obtenemos la posición en pantalla actual del zombie y dibujamos la baba en él.

3. Vemos a donde nos tenemos que desplazar y copiamos las páginas de forma correcta. También modificamos los valores de x ó y del zombie según corresponda:

$mmu\_copiar\_pagina((unsignedint*)0x8000000, (unsignedint*)0x8005000);$
$zombie\_actual.y = zombie\_actual.y + izquierda;$
Algorithm 1: Caso de izquierda

$mmu\_copiar\_pagina((unsignedint*)0x8000000, (unsignedint*)0x8004000);$
$zombie\_actual.y = zombie\_actual.y + derecha;$
Algorithm 2: Caso de derecha

$mmu\_copiar\_pagina((unsignedint*)0x8000000, (unsignedint*)0x8001000);$
$zombie\_actual.x = zombie\_actual.x + adelante;$
Algorithm 3: Caso de adelante

$mmu\_copiar\_pagina((unsignedint*)0x8000000, (unsignedint*)0x8006000);$
$zombie\_actual.x = zombie\_actual.x + atras;$
Algorithm 4: Caso de atras

4. Dibujamos el zombie usando los nuevos x e y.

5. Revisamos si el valor de x es 0 o 77. Ya que en estos dos casos significa que el zombie a llegado al área de uno de los jugadores, el primero en el caso de 0 y el segundo en el caso de 77. En estos casos se debe aumentar el puntaje del jugador que corresponda e imprimirlo. Además se debe matar a la tarea zombie actual.

6. Remapear

mmu_copiar_pagina copia de dirección lógica a dirección lógica una página entera. Los valores de origen y destino se especifican en el enunciado.

8.6. Debug

Para poder completar el modo debug lo que hacemos es agregar las siguientes variables globales:

1. Una variable booleana debug_on que cuando sea true nos indicará que el modo debugger está activado, desactivado en el caso contrario.

2. Una estructura datos_debug. Esta contendrá la información necesaria para mostrar en el debugger. Lo que contendrá será lo siguiente: eax, ebx, ecx, edx, esi, edi, ebp, esp, eip, cs, ss, ds, fs, gs, es y eflags.

3. Un arreglo de short de tamaño 4000 para respaldar la pantalla de nombre respaldo_pantalla.

4. Otro booleano de nombre pantalla_debug_activa que dirá si la pantalla de debug se está mostrando ó no.

5. Un último booleano de nombre no_jump que es usado solamente dentro de la rutina de atención de excepciones para poder facilitar el código.

Agregamos la letra 'Y' en el convertir_scancode y lo que hará sencillamente es invertir el valor de debug_on en el caso que pantalla_debug_activa este en falso. En el caso contrario, pintará la pantalla para que quede igual a como era antes de caer la excepción y continuará las tareas. Para esto último solamente

pondremos un condicional en la rutina de atención de interrupción de reloj para evitar que salte a otra tarea si la pantalla de debug está activada.

Utilizaremos una función de nombre debug_zombie que se encargará de dibujar en la pantalla, previamente respaldada y los datos que pide el ejercicio. Modificaremos la macro que utilizamos para atender las excepciones para que, si debug_on está activado, completará datos_debug y llamará a debug_zombie. También se debió tener en cuenta cuales de las excepciones devuelven el código de error y cuales no. Esto tiene que ver con como se obtienen los datos. Muchos de los cuales debemos conseguir de la pila como por ejemplo el cs ó eip.

8.7. Juego

Para el juego agregaremos dos archivos game.c y game.h. Estos nos servirán para moldear el movimiento de los jugadores, el cambio de tipo de zombies y su lanzamiento. Para esto usaremos las siguientes variables:

1. Un entero jugA_pos que sirve para indicar en que posición de la columna roja esta el lanzador del primer jugador. Su valor predeterminado es el 0 (parte superior de la columna).

2. Un entero jugB_pos ídem anterior para el segundo jugador.

3. Un entero jugA_ztype que indica que tipo de zombie tiene el lanzador actual. Si es 0 entonces es guerrero, 1 es mago y 2 es clérigo. Su valor predeterminado es el 0 (tipo guerrero).

4. Un entero jugB_ztype ídem anterior para el segundo jugador.

Usando la función game_inicializar podremos asignar los valores predeterminados a estas variables. Además de inicializar el sched y los jugadores, pintarlos en pantalla y dibujar el puntaje y los zombies restantes de cada uno.

Para lanzar un zombie utilizaremos la función game_lanzar_zombie. Esta lo que hará es, si hay un zombie disponible, marcarlo como vivo. Recibirá como parámetro el jugador con el que deberá realizar esta acción. Además debemos asignarle la posición adecuada a la estructura zombie(su x e y). Completaremos su tss usando completar_tss_zombie que describimos en el punto 6. Mapearemos las páginas del zombie en las posiciones adecuadas usando game_cargar_codigo_zombie. Lo pintaremos en pantalla e incrementaremos la cantidad de zombies lanzados del jugador, recordar que solamente puede tener 8 en juego. Y por último pintaremos este dato.

Como aclaración, para que exista un zombie disponible se tiene que cumplir que alguno de los zombies del jugador este marcado como muerto y además el jugador debe hacer lanzado menos de 20 zombies.

También modificamos la función convertir_scancode para que cuando se oprima cada una de las teclas se realiza la acción apropiada del juego. En el caso de que la pantalla del debug este activa, la única tecla que hará algo sera la 'Y' que lo desactivará.

1. En el caso de apretar 'A' ó 'D' el zombie del jugador uno cambiara de tipo, alternando entre guerrero, mago ó clérigo(Cambia el valor de jugA_ztype).

2. En caso de 'W' y 'S' el lanzador se moverá arriba y abajo respectivamente(Cambia el valor de jugA_pos).

3. En el caso de la tecla shift izquierdo lo que hará es lanzar el zombie pasándole como parámetro a la función game_lanzar_zombie un 0(jugador 1).

Estos procesos se hacen sencillamente cambiando el valor de las variables descriptas al principio de la sección, según corresponda. Para las teclas 'I', 'K', 'J', 'L' y shift derecho se hace lo mismo pero para el segundo jugador y sus respectivas variables.