DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Ramiro Ciruzzi	228/17	ramiro.ciruzzi@gmail.com
Oscar Alvarez	619/18	oscar_algu@hotmail.com
Luciano Melhem	197/20	lucianomelhem@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

Índice

0. Introducción

En el siguiente trabajo práctico se ponen en práctica los conceptos de System Programming con un juego de Lemmings. A lo largo del mismo iremos resolviendo ejercicios gradualmente hasta llegar a una implementación funcional del juego que corre en el simulador de S.O. bochs.

1. Ejercicio 1

1.1. Armado de GDT

Definimos en la GDT los descriptores de índices 8,9,10,11 que corresponden a código de nivel 0, código de nivel 3, datos de nivel 0 y datos de nivel 3 respectivamente. Estos descriptores difieren entre sí únicamente en dos campos: el campo *Type* y el campo *DPL*.

Para los descriptores de código *Type=0x0A* ya que son de código Execute/Read y para los descriptores de datos *Type=0x02* ya que corresponden a datos Read/Write. En cuanto al *DPL* aquellos de nivel 3 tienen *DPL=3* y los de nivel 0 tienen *DPL=0*.

Luego para el resto de los campos estos descriptores tienen todos lo mismo:

- *Base=0* porque deben direccionar los primero 817MiB. Como la base es 0 nos queda segmentación flat.
- *S=1* ya que son segmentos de código o datos.
- *Limit=0x330FF* ya que si queremos direccionar los primeros 817MiB esa debe ser la dirección del último bloque de 4KiB direccionable.
- *G=1* ya que queremos que el límite se interprete en bloques de 4KiB.
- *AVL=0* ya que no lo usamos.
- *P=1* para marcarlos como presentes.
- *L=0* porque son segmentos de código o datos de 32 bits.
- *D/B=1* siempre en 1 para segmentos de código o datos de 32 bits.

1.2. Pasaje a modo protegido y seteo de pila del kernel

Para pasar a modo protegido habilitamos el address a partir de la dirección 20, cargamos la GDT en GDTR para tener segmentos disponibles en los cuales ejecutar código y cargar datos. Activamos el bit menos significativo de CR0 que habilita el modo protegido y finalmente saltamos con el selector de segmento con el que se ejecutará el código de nivel 0.

Una vez en modo protegido, seteamos la pila del kernel asignándole 0x25000 al registro ESP. También seteamos el registro de segmento de stack como uno de datos nivel 0.

1.3. Creación segmento de vídeo

Para crear un nuevo segmento completamos los campos de la entrada 12 de la GDT con lo necesario. Veamos los campos más significativos, aquellos no descriptos aquí abajo son iguales a los de los descriptores definidos en la sección 1.1.

- *Base=0xB8000* ya que es donde empieza la pantalla en memoria.
- *Limit=0x1F3F* ya que la pantalla es de $80x50x2$ (x2 es por byte de carácter y otro de atributo) y apunta al último byte direccionable.
- *G=0* ya que queremos que el límite se interprete en bloques de 1byte.
- *DPL=0* porque el segmento de video debe ser accesible solo por el kernel.

- *Type=0x02* ya que corresponden a datos Read/Write.

Con esto, al acceder con el selector de segmento de video correspondiente a la nueva entrada de la GDT ($GDT\_IDX\_DS\_RING\_0\_VIDEO << 3$, con $RPL = 0$ porque se accede a un descriptor con $DPL = 0$, y $TI = 0$ porque el selector se refiere a un descriptor de la GDT) para pintar la pantalla, ya no necesitamos indicar que queremos escribir a partir de la dirección $0xB8000$. Con este selector ya podemos apuntar sin offset a esa memoria, y además, nos aseguramos de no pasarnos y escribir en otra parte gracias al limite del descriptor. Para hacer esto, llenamos un registro de selector de segmento GS con el selector de video, y luego vamos accediendo con ese selector a memoria con offsets en el rango $[0; 0x1F3F]$ para pintar la pantalla.

1.4. Usando asm para limpiar y pintar la pantalla con un color de fondo, junto con las barras de los jugadores

Hicimos una modificación a la macro brindada por la cátedra $print\_text\_pm$ que consiste en acceder a la memoria de video con el nuevo segmento utilizando la dirección lógica *selectorVideo:offset*, aumentando ese offset de 2 en 2 y llenándolo con el carácter del string y el atributo (color) pedido. El selector de video es el que apunta al segmento de video que creamos en la GDT, y el offset empieza de cero. Para printear una pantalla vacía, creamos un string de 4000 bytes con caracteres vacíos y lo printeamos con la macro.

Para pintar la pantalla hicimos otra macro *pintarPantalla*. Esta usa la macro anterior, y va pintando cada figura de la pantalla con los offset y atributos necesarios. Pinta en verde, pone los números para los clocks, los nombres de los equipos y los paneles con su conteo de spawneados.

2. Ejercicio 2

2.1. Completar la IDT para las excepciones del procesador. Imprimir en pantalla la excepción. Desalojar la tarea que produjo la excepción.

Las excepciones del procesador deben solo poder ser invocadas directamente por el procesador, y de ser ejecutadas manualmente con una instrucción *int*, se debe hacer desde nivel cero. Por esto, el DPL del descriptor de la interrupción debe ser 0. Además, el RPL del registro del selector cargado también debe ser 0, ya que para poder acceder manualmente a una interrupción la protección que se aplica es la de que el EPL debe ser menor igual numéricamente al DPL.

En cuanto al código de la excepción, se ejecuta en nivel 0, por lo tanto en la IDT va cargado con el selector de segmento de código de nivel 0. Y la dirección de donde comienza el handler se carga también en la IDT, con la etiqueta del comienzo de la función ubicada en el kernel.

Para definir el código de todas las excepciones, tenemos muchas definiciones de funciones creadas a partir de una macro ($\_isr\,\%1$). Cada id de excepción se corresponde con su respectiva excepción. En caso de que este id de excepción valga entre 0 y 21, sin incluir el 15 (reservada por intel), se pushea a la pila el id de la excepción para que sea printeado por una función en C, *imprimirExcepcion*. Esta función, simplemente printea la descripción y el numero de la excepción.

Por ahora, dejamos un loop infinito $jmp\$$ al final del handler para que no se siga produciendo el error. Mas adelante se hará una rutina que desalojará la tarea.

2.2. Generando excepción de prueba

Para poder utilizar la IDT, primero creamos las entradas en la IDT en tiempo de ejecución con la función idt_init y luego cargamos la tabla con la instrucción $lidt$ (con la etiqueta que apunta a la estructura de la IDT ubicada en el kernel). Posteriormente realizamos una división por cero en *kernel.asm* y verificamos, con un breakpoint ubicado en el código de las excepciones si llega. Para ver que efectivamente sea una división por cero, vemos el parámetro *%1* del id de la excepción que le pasamos a la función *imprimirExcepcion* desde la pila (*x/1wx esp* justo antes de ejecutar *imprimirExcepcion*). Luego vemos que llegue a $jmp\$$.

3. Ejercicio 3

3.1. Agregando a IDT entradas para interrupción de reloj, teclado y para interrupciones de software de servicios

El DPL y CS de las interrupciones de teclado y reloj se cargan con los mismos valores que las excepciones del procesador por las mismas razones: la interrupción no se puede invocar con *int x* manualmente desde código de nivel 3 (DPL=0) y se ejecuta desde nivel cero (CS=0).

Por otro lado, como las otras interrupciones van a estar asociadas a los servicios (88,98,108) tienen que poder invocarse manualmente desde las tareas, osea desde código de nivel 3. Esto se ve reflejado en el DPL de los descriptores, que es de nivel 3. En este caso, el RPL del selector de segmento de código en ejecución no necesita verificar ninguna condición (puede ser cero o tres) para poder acceder a un descriptor con el menor nivel de privilegio posible.

Por último, para todas estas interrupciones, el offset del descriptor se completa con la dirección donde esta el handler. Esta dirección coincide con la de la etiqueta del handler ya que hay segmentación flatt.

3.2. Escribir la rutina asociada a la interrupción de reloj que muestra el reloj del sistema rotando

Definimos la interrupción 32 para el reloj. Esta debe llamar a la función *picFinish1* para avisarle al controlador p̈rogrammable interruption controllerde interrupciones externas que ya se atendió la interrupción y que una vez que termine se pueda interrumpir la ejecución con otras interrupciones externas. Para imprimir en pantalla el clock del sistema, se hace un call a *nextClock* antes de iret (en el handler del reloj). También se pushean todos los registros y se popean, para que la interrupción sea transparente.

3.3. Rutina de interrupción de teclado, con impresión de números en la esquina derecha y scancodes en centro de la pantalla.

También definimos la interrupción 33 para el teclado. Esta al igual que la anterior debe hacer un llamado a *picfinish1* para desenmascarar las demás interrupciones externas una vez que termine de interrumpir la actual. La interrupción de teclado recibe un s̈can codeq̈ue mapea cada tecla a un número de 1 byte. Con la instrucción ïnp̈odemos leer el buffer del teclado, osea el scancode de la tecla, y pasarlo a un registro.

En caso de que la tecla sea un número, esta estaría por arriba del scancode 0xB, por lo tanto hacemos un *jump* (sin signo) para imprimir la tecla en el centro de la pantalla. En caso contrario, llamamos a una función desde C que printea el numero en la esquina superior de la pantalla, osea en la fila 0 y la columna 79. Transformamos el scancode a número restándole uno y lo printeamos. Cabe destacar que el cero es un caso especial porque tiene otro scancode. Una vez que printeo el número o el scancode de la tecla, termina con iret.

3.4. Escribir las rutinas de las interrupciones 88, 98 y 108 para modificar eax

Dentro del handler, para modificar el valor de eax, simplemente se mueve a eax un inmediato, sin pushear registros en la pila. Esto haría que la instrucción por el momento no sea transparente.

4. Ejercicio 4

4.1. Inicialización de directorio y tablas de pagina del kernel

Para hacer el mapeo del kernel lo primero que hacemos es inicializar 2 punteros en las direcciones especificas para ubicar el directorio y la tabla de pagina del kernel (0x25000 y 0x26000). Luego limpiamos los directory index y table index. Los de directorio para que no hayan descriptores inválidos o que casualmente mapeen memoria que no queremos, y los da tabla de pagina para que la tabla que mapeamos

por completo no tenga basura en los bits de accessed (que se prende en uno cuando leemos/escribimos), dirty (que se prende en uno cuando escribimos), entre otros bits.

Para el page directory y las tablas se completan algunos bits de importancia. Uno de ellos es el bit de presente que indica si la pagina esta cargada en la RAM y no hay que ir a buscarla a disco. Otro es el bit de privilegio, que lo seteamos en 0 porque es el kernel. Y el read/write, en 1 porque queremos escribir datos en el kernel, por mas de que haya regiones de código (las regiones de código no se pueden escribir, esto se bloquea con segmentación si se usa el segmento de código).

En particular, para el page directory entry, colocamos en la dirección base los 20 bits mas significativos la dirección de la tabla de pagina que creamos $(0x26000 >> 12)$. Y con respecto a la tabla, la llenamos por completo, ya que cada entrada mapea un marco de pagina de 4kib y hay 1024 entradas, sumando un total de $4kib * 1024 = 4mib$ de espacio accesible gracias al mapeo (que son las 0x400 000 direcciones pedidas). Cada base (dirección física a la que mapea) va a valer 0,1,2... ya que representan las direcciones físicas que se obtienen al colocar paginas desde la dirección cero, shifteadas 12 a la derecha $(0x0000- > 0, 0x1000- > 1, 0x2000- > 2...)$.

Finalmente devolvemos la dirección del page directory para que pueda ser cargada en el cr3 y que el procesador tenga acceso al kernel una vez activada paginación.

4.2. Activar paginación

Para activar paginación, primero debemos cargar el cr3 con el directorio creado al inicializar el mapeo del kernel, para que esta sea la visión que tiene el procesador de la memoria al activar paginación. Una vez que el cr3 esta cargado, activamos paginación modificando el bit mas significativo (Paging) del registro de control cr0 (movemos su contenido original a un registro de propósito general, le ponemos un 1 arriba, y lo volvemos a cargar en cr0).

4.3. c)Escribir una rutina que imprima el numero de libreta

Lo que hacemos es usar las funciones provistas de print que acceden al buffer de memoria de video y printean caracteres con su color de fondo y de letra en la posición de la pantalla indicada.

5. Ejercicio 5

5.1. *mmu_init*, inicialización de estructuras (variables) para administrar la memoria

La memoria del sistema se administra con dos contadores: uno para paginas libres de nivel cero y otro de nivel 3. El de paginas libres de nivel cero empieza en la dirección 0x100000 y esta dentro del kernel. Este después de iniciar el juego tiene paginas libres, page directories/tables y pila de nivel cero. El área de usuario, en cambio, empieza en la 0x400000 y almacena pilas de nivel 3 (tareas) y memoria compartida. La idea es ir sumando de a una pagina cada vez que se pide memoria del área libre del kernel con la funcion *next_free_kernel_page*, o del área libre de usuario con su función análoga. A medida que se piden paginas, se van incrementando, de a una pagina, las variables globales que apuntan a la próxima pagina libre de lo que sea que se haya pedido, kernel o usuario.

En caso de que los contadores lleguen a su limite (0x400 000 para kernel y usuario 0x3000 000) se agotan las paginas libres. Si se pidiesen muchas paginas de usuario se van a dar paginas que están fuera del rango previsto para la direcciones físicas. Cuando el sistema funciona normalmente no se deberían pedir paginas para direcciones físicas de más. Para usuario hay 44mib, pero solo se usan 32Mib+4kib*10 de memoria como maximo (memoria compartida y pilas de nivel 3), nos sobran las que van de 0x2 40a 000 a 0x3 000 000. El kernel guarda page tables, page directories y pilas de nivel cero. Las pilas son siempre 10, y cuando mueren los lemming se van usando las mismas. En cambio, al mapear, la cantidad de page tables/directories puede ir aumentando. Como máximo, obtendremos 10 pd y 1024*10 tablas que ocupan 4kib cada uno, por un total de 40Mib+40kib para mapear 4 GB. No alcanzaría todo el tamaño del área libre del kernel para guardar todas estas pd/pt (ya que tiene 3 Mib). Sinembargo, no se

requerirían jamas, porque solo necesitamos una cantidad máxima de 8 paginas de pd/pt para cada tarea. 1 pagina para la tabla que mapea el código y la pila de nivel 3, una para el kernel, 4 a lo sumo para memoria compartida, otra para el page directory y una ultima para la pila de nivel cero.

5.2. Mapeo *mmu_map_page*

Cuando se traduce una dirección lineal, en 32 bit sin PAE, al menos se necesita acceder a las entradas en page directory y page table correspondientes a la dirección lineal. Mas detalle de traducciones ver 10.1. En caso de que queramos mapear una pagina física a esa dirección lineal, usaríamos esta función. Lo primero que hace esta función es extraer de la direccion lineal los campos directory y table. Luego, para tener acceso a la base del page directory, se limpian los primeros 3 nibles del cr3, que contiene la dirección lineal en donde se encuentra el page directory (que en este caso coincide con la física porque esta en el kernel, que tiene identity mapping).

Una vez hecho esto, se accede a la entrada del page directory correspondiente al campo directory de la direcciona lineal que se quiere mapear. En caso de que este presente, podemos afirmar que ya hay un page table definido para esa entrada, por lo tanto solo resta completar en esa tabla la entrada que le corresponde al campo table de la dirección lineal. Si no, se define un nuevo page table: se llenan todas sus entradas con cero para evitar que queden mapeos no deseados por basura que haya quedado en las direcciones de memoria donde definimos la PTE. Y se rellena la entrada del page directory con todos los permisos en 1 y la dirección de la nueva tabla (shifteada 12 porque los demas bits se utilizan para permisos). Los 12 bits que no se llenan de esta dirección física son utilizados para los permisos en el descriptor (page directory entry/page table entry).

Los permisos del directory entry van en 1 porque lo que termina quedando es un AND entre los permisos del directory y del la tabla. De poner los permisos de una entrada de directorio en cero, ninguna de las paginas mapeadas por la tabla a la que corresponde se podrían mapear con nivel usuario o con permisos de escritura, considerando que CRO.WP=1. En nuestra implementación no pusimos este bit en 1, ya que todos los mapeos de nivel cero son read/write. En caso de que se haga un mapeo con nivel supervisor y readOnly, se podría escribir. Para evitar esto se debería prender el bit.

Finalmente, llenamos la page table entry correspondiente a la dirección lineal con los bits de permisos que recibe esta función, y la dirección física en la que se encuentra el marco de pagina, shifteada 12 ya que los primeros 12 bits de la dirección son cero y no necesitamos guardarlos.

5.3. mmu_unmap_page

A diferencia de la función anterior, no necesita recibir una dirección física, ya que de estar mapeada se conoce la dirección física, y tampoco tiene relevancia. En caso de no estar mapeada, unmap devuelve cero y no hace nada. De estarlo devuelve la dirección física que desmapeo, y la obtiene de la tabla de pagina correspondiente a la dirección lineal recibida.

Realizamos el recorrido de la traducción chequeando los bits de presente, primero del entry y luego de la tabla. De estar presente los dos, pone en cero el de la tabla (para no desmapear otros marcos de pagina que contenga ese directory entry). Con esto, la memoria queda inaccesible hasta que se realice otro mapeo. Para asegurarnos que los cambios de desmapeo sean efectivos, necesitamos limpiar la cache de traducciones (tlb). Si no se hiciera esto, al desmapear una pagina la cache seguiría teniendo su traducción y se podría seguir accediendo a las direcciones de memoria de ese mapeo.

5.4. mmu_init_task_dir

Para crear un mapeo para la tarea, primero pedimos una pagina del area libre del kernel para guardar el page directory. Luego le llenamos las 1024 entradas todas en cero para que no quede algún bit de presente prendido con una dirección de page table invalida (Que podría generar un page fault si la accedemos).

Lo siguiente, mapeamos el kernel para que: puedan caer interrupciones, todas las direcciones lineales se puedan traducir (tengan acceso a los page directory, paga tables de la tarea que están en el kernel) y se pueda tener acceso a la pila de nivel cero (que sale del área libre del kernel). Las interrupciones

no podrían caer si no esta el kernel mapeado porque generaría un page fault cuando trata de saltar al handler, código situado en el kernel. Por esto mismo, el cr3 de cada tarea necesita tener un mapeo del kernel para poder ser interrumpido.

Mas adelante en la función, nos encargamos de mapear el código de la tarea y la pila. Para esto usamos la función anterior *mmu_map_page*. En el caso del código de la tarea, al mapeo le asignamos atributos de solo lectura y de usuario. La dirección física es la recibida por *mmu_init_task_dir* y la dirección lineal es la 0x8000000. Para la siguiente pagina de código se suma una pagina a la dirección física y virtual, es decir, la consecutiva. La pila también tiene una dirección lineal definida por el enunciado, pero esta mapea a una física que es la siguiente pagina libre de usuario (en nuestro caso $0x400000 - 0x40a000$, de la memoria de tareas). En cuanto a los permisos, la pila puede leerse y escribirse, y es de nivel 3.

Finalmente la función devuelve la dirección del directorio de tabla de pagina para que pueda ser cargado en el cr3 en la tss, y la tarea tenga acceso a todo este mapeo.

5.5. Construir mapa de memoria para tareas e intercambiarlo por el del kernel. Escribir el primer carácter de la pantalla, mirar los mapeos actuales y volver a la normalidad

Para intercambiar el mapa de memoria del kernel, llamamos a la función que creamos para iniciar el mapeo de una tarea y cambiamos el cr3 por el que devuelve esta función. Para que la escritura de la memoria de video funcione, deberíamos haber mapeado el kernel en el cr3. Este test verifica eso. Luego, al hacer info tab podemos apreciar los 3 mapeos: el del kernel que mapea con identity mapping los primeros 4Mib, el de la pila de nivel 3 que va desde el rango de lineales $[0x802000; 0x803000]$ a la siguiente pagina libre de usuario osea $[0x400000; 0x401000]$, el del código que va desde $[0x800000; 0x802000]$ al kernel osea al rango de direcciones $[0x18000; 0x1a000]$(amalin) o $[0x1a000; 0x1c000]$ (betarote).

6. Ejercicio 6

6.1. Definición de descriptores de TSS en la GDT para tarea inicial y tarea idle. Completado de descriptores

Los descriptores de TSS son descriptores de tipo sistema (bit en 0), no necesariamente definen un segmento. Estos apuntan a la dirección lineal donde esta definida la TSS, que de no ser modificada, contiene el ultimo contexto de ejecución guardado por el procesador al terminar de ejecutar la tarea asociada a esa TSS y saltar a otra.

Los descriptores se ubican en el siguiente índice disponible en la GDT. Cada descriptor tiene 4 campos en los que hacemos incapie:

- Type que contiene el bit de busy. El bit de busy indica que tarea se esta ejecutando y lo va completando el procesador mientras va intercambiando tareas.
- La base es esa dirección que apunta a la TSS, osea el puntero a *tss_idle/tss_inicial*
- El limite es ultimo byte direccionable de la TSS, es decir el tamaño en bytes menos uno (0x67).
- Y el DPL indica que EPL (max(CPL,RPL)) se necesitan para poder intercambiar a esa tarea.

El CPL, en nuestro caso siempre es cero porque saltamos desde una interrupción. Nos tenemos que asegurar que el RPL sea 0 también, en caso de que el descriptor de TSS de la tarea tenga un DPL=0. Los lemmings, por ejemplo, tienen DPL=0 para que estos no puedan saltar uno al otro desde el código de nivel 3. Tienen que pasar por una interrupción, ejecutada en nivel 0, para poder hacer el intercambio.

La TSS de tarea inicial, como no ejecuta nada, se llena en cero. El único propósito de esta tarea es que el procesador tenga una TSS en donde guardar el contexto actual. El descriptor es uno de TSS (con los bits de tipo correspondientes y el limite indicado), pero no utilizamos ese contexto para nada porque una vez que cargamos el TR de la inicial lo único que hacemos es saltar a la idle.

Para la TSS de la tarea idle, en cambio, si llenamos su descriptor. La base se inicializa en ejecución con la *funcion tss_init*, ya que la tss se carga en la memoria del kernel al iniciar el sistema. Antes del inicio no sabemos la dirección en donde esta cargada la tss. Lo mismo ocurría con las etiquetas de los handlers de interrupción, se cargaban en ejecución. Los demas campos de la GDT se llenan igual que la tarea inicial. La TSS de la tarea idle comienza con:

- registros de propósito general en cero
- el esp en la dirección del tope del stack del kernel indicada en el enunciado (asumimos que no volvemos al código de kernel.asm, por lo tanto no nos interesa lo que quedo cargado en la pila, si hubiese algo)
- esp0 y ss0 no es necesario inicializarlos porque la idle corre con cs de nivel cero y no va a generar un cambio de privilegio cuando salte a alguna interrupción de nivel cero (lo mismo para esp1,ss1,esp2,ss2)
- el cr3 lo cargamos con el del kernel, de enunciado.
- el eip con la dirección donde empieza el código de la tarea (0x1c000)
- los flags con las interrupciones habilitadas para poder salir de la tarea con la interrupción del reloj/teclado
- los selectores de segmento con código y datos de nivel 0 para que se puedan efectuar todas las instrucciones que los tengan implícitos (push para ss,mov ds, etc.), y para que el código de la idle corra en nivel cero.
- iomap en 0xffff, para no utilizar entradas en la TSS relacionadas con dispositivos E/S

Finalmente, como la tarea idle esta en el kernel, y tiene cargado el cr3 del kernel, ya nos aseguramos que su código este mapeado con identity mapping.

6.2. Saltar entre idle y inicial

Para cargarle un contexto inicial al procesador, cargamos el selector de la tarea inicial en el TR con la instrucción ltr, pasandole como operando dicho selector (el definido en la GDT).

Antes de hacer un jmp far con el selector de TSS de la tarea idle, con tss_init completamos lo que faltaba del descriptor de tss: la base del descriptor de tss de la idle.

6.3. Inicializar una tarea

Creamos un arreglo de *tss tss_tasks* para guardar el contexto de cada una de las 10 tareas de los lemmings. La función que creamos fue *tss_task_init*, que recibe un nro de tarea y un page directory. Lo primero que hacemos en esta función es llenar una entrada del arreglo de tss con lo que llamamos un *tss_limpio*. Que tiene los siguientes campos genéricos para las tareas de nivel 3:

- selectores de nivel 3
- eip apuntando a la direccion lineal del codigo (0x8000 000),
- esp apuntando al tope pila del codigo (0x8003 000)
- ss0 de datos nivel cero
- eflags con las interrupciones habilitadas.

Luego cargamos el cr3 según lo pasado por parámetro (page directory), que esperamos que ya venga con el mapeo hecho por *mmu_init_task_dir*. Esta función de inicio de TSS siempre debe ser llamada después de la que crea el mapeo, para poder cargar el cr3. Tambien llenamos el esp0 (porque acá si pueden ocurrir cambios de privilegio) con una nueva pagina libre del kernel, y lo hacemos apuntar al final de esta. Y llenamos la base del descriptor de la TSS con la dirección de la entrada del arreglo *tss_tasks*.

7. Ejercicio 7

7.1. Inicializar las estructuras de datos del scheduler

Para inicializar las estructuras que necesita el scheduler (las TSSs y los mapeos) creamos una primer entrada en la GDT (indice 15) que es la base de las tareas, y replicamos ese descriptor de TSS 10 veces en índices posteriores. Este descriptor es igual al de la idle.

Vamos pidiendo un mapeo para cada tarea. Definimos los números de Tarea (de 0 a 9) pares como identificadores del equipo amalin y a los impares como identificadores de betarote. Los identificadores, a su vez, coinciden con el índice en el arreglo de TSSs. Para cada equipo mapeamos a la dirección física pedida. El mapeo hecho se lo pasamos a *tss_task_init* junto con el numero de la tarea para que inicie la TSS de la tarea.

Con respecto a los descriptores de la GDT, los nrosTarea quedan corridos:

$$indiceGDTdeTSS = nroTarea + GDT\_IDX\_TSS\_TAREAS\_BASE.$$

Ejemplo: la tarea nro 3 (contando desde cero, la segunda betarote) tiene su descriptor definido en el índice 18 de la GDT. De ahora en mas, para referirnos a la tarea que representa al lemming en juego nos referimos al 'lemming' directamente.

7.2. El scheduler, una función que devuelve el índice de la siguiente tarea a ejecutar.

Como invariante, nroTareaActual nos dice la ultima tarea que fue ejecutada de cada equipo. Para saber que lemmings están vivos y hace cuanto están vivos, introducimos un arreglo de longevidad. El arreglo de longevidad empieza con todos ceros (todos muertos). Cada elemento tiene un índice que se corresponde con el nro de tarea y un valor que se corresponde con la longevidad. Numéricamente, a mayor longevidad, mas joven es el lemming. Y el leming con menor longevidad es el mas antiguo.

Para devolver el índice de la siguiente tarea a ejecutar, primero buscamos el siguiente vivo (*longevidad[nroTarea]* $\neq$ *cero*) del equipo contrario en el orden que aparece en el arreglo longevidad (primer while en *sched_next_task*). En caso de estar, devolvemos el indice en la gdt correspondiente a la TSS de esa tarea. Sino (segundo while de *sched_next_task*) seguimos buscando en el equipo que acaba de terminar su turno mientras haya otro lemming vivo que no sea el que se esta ejecutando (*nroTarea[equipoJugando]+=2*). En caso de que hubiera uno, devolvemos ese. Sino, devolvemos el índice de la única tarea que se esta ejecutando. Desde la interrupción de reloj contemplamos este único caso, y no saltamos (intercambiamos de tarea) si el scheduler devuelve la misma tarea que se esta ejecutando (es decir, es el ultimo lemming vivo). Y si están todos muertos, el scheduler devuelve el indice de la idle en la GDT para saltar ahí (o no hacerlo en caso de que ya este en la idle). Esta tarea idle se mantiene en ejecución con la misma lógica con la que se mantiene en ejecución el único lemming vivo.

Al final de la función, para devolver el índice del descriptor de TSS en la GDT (cuando no están todos muertos) partimos de una etiqueta base (*GDT_IDX_TSS_TAREAS_BASE*) y devolvemos esa base mas el numero de tarea. Los índices debajo de esa base son los descriptores de la tarea idle o de la tarea inicial, los reservados, o los descriptores de segmento. Los que están por encima son los de las tareas.

Cabe destacar que el scheduler solo devuelve lemmings vivos, es decir índices en la GDT de descriptores de TSSs cuya longevidad es distinta de cero. Osea que no se va a saltar a ninguna tarea que tenga longevidad cero.

Chequeo y conteo de ciclos

Introducimos el chequeo y conteo de ciclos en el scheduler, porque tiene que ver con los turnos de los Lemming. Guardamos en una variable la cantidad de ciclos que se ejecutan, que vamos incrementando cada vez que se ejecuta el scheduler. El chequeo de ciclos lee esta variable y cada determinada cantidad de ciclos hace nacer lemmings (*nacerSiguiente*) y mata/revive al mas longevo (*matarYrevivirViejos*).

No tendría sentido, por ejemplo, seguir contando ciclos cuando se esta con el debug trabado ni cuando ya termino el juego. Por esto decidimos no ejecutar el conteo y chequeo de ciclos en esos 2 casos. Al estar

en el scheduler, esto ultimo ocurre sin ninguna modificación, ya que el scheduler no se ejecuta en ninguno de estos 2 modos.

Un detalle es que guardamos el equipo actual pero que podríamos haberlo obtenido traduciendo el TR (*equipoActual=trAnroTarea() %2*).

Aclaraciones con respecto al enunciado:

1) Ejecutamos uno después del otro como aparece en el arreglo. Como lo recorremos circularmente, si nace uno siempre vamos a pasar por este antes de volver a ejecutar la tarea con la que comenzamos, obviamente a menos que esta muera en el proceso. Es decir, deben pasar por todas las demás tareas (las que están vivas) para volver a la actual. Por lo tanto todas las tareas son ejecutadas al menos una vez antes de volver a la misma.

2) Podemos comprobar que con esta función *sched_next_task* no se ejecutan nunca dos tareas seguidas del mismo pueblo a menos que sea la única viva entre los 2 equipos.

7.3. Realizar el intercambio de tareas para que ejecuten lemming A y B en cada ciclo de reloj

Inicialmente, para que vivan les ponemos un numero distinto de cero en longevidad a los lemming 0 y 1, osea en los índices 0 y 1 (mas adelante van a nacer solos). Modificamos la rutina de reloj para que ejecute *sched_next_task* y convierta el índice a selector y en caso de que sea distinto al del TR intercambie tareas, sino haga iret. Para verificar que los lemming están jugando le ponemos un breakpoint en el código de la tarea y hacemos info tss cuando cae. Así vemos que el selector de segmento se corresponde con un lemming de A y luego con un lemming de B.

Para **escribir el reloj de cada lemming**, nos basamos en la función que imprime el reloj del sistema. Esta función guarda un estado del clock de cada tarea en la memoria del kernel con la etiqueta *baseTaskClockState* y accedemos a la misma con un offset de 4 * nroTarea bytes. Con esto nos aseguramos que cada tarea tenga un clock especifico. Osea que por ejemplo para la tarea 3 el estado del clock (offset en el string de símbolos) esta determinado por DWORD [baseTaskClockState + 4*3]. Con este offset accedemos al string del clock "$isrClock$" y printeamos el símbolo. Para printearlo en el lugar indicado, hacemos un código para convertir el nroTarea en el offset de la columna. Este offset se calcula en los bloques de código debajo de las etiquetas *next_clock.amalin* y *next_clock.betarote* (*offsetColClock=(nroTarea-1)*2* para betarote y *offsetColClock=nroTarea*2* para amalin).

Como decisión extra, **el clock siempre se imprime cuando se entra al código de una tarea**, por mas de que la tarea sea desalojada inmediatamente. Para esto la impresión del clock se encuentra antes de cualquier intercambio de tarea. Por ejemplo, al saltar a la idle desde el move o al desalojar una tarea por explotar, bridge, una excepción del procesador, o al pasar a la siguiente tarea mediante el scheduler.

7.4. Modificar move (int88) para que salte a la idle, y verificar que su reloj se ejecuta

Para que salte a la idle hacemos un jmp con el selector de TSS de la idle desde la rutina de interrupcion 88. El RPL es cero porque queremos ejecutar una tarea con DPL=0. Para verificar que se ejecuta, dejamos un syscall move en el código de un lemming y ponemos un breakpoint en la interrupcion 88. Hacemos continue y vemos como se salta a la idle, y seguimos con next hasta ver que el reloj de la idle gira.

7.5. Desalojar la tarea cuando hace una excepción

Para desalojar tarea utilizamos 2 funciones que no explicamos todavía, y un jmp a la idle. Una función, trAnroTarea, nos devuelve el numero de tarea según el TR que esta cargado actualmente en el procesador. Esto lo necesitamos para matar al lemming. Y la otra función que lo mata y desmapea recibe ese numero de tarea, y con este pone en cero su valor en longevidad.

Algo a tener en cuenta es que al lemming en pantalla lo quitamos cuando ocurre la excepción, antes de la macro desalojar tarea. Lo quitamos reemplazando el casillero donde estaba con lo que había antes (esto de dejar lo que había antes se explica mas adelante, en explode). Esta decisión de separarlo de

la macro fue porque hay otras funciones que necesitan esta funcionalidad de desalojar la tarea pero ya destruyen al lemming de pantalla antes.

Para nosotros, saltar a la idle implica perder el turno del lemming, osea esperar a que se ejecute la siguiente interrupción de reloj y que el scheduler ejecute el siguiente lemming del arreglo.

7.6. Modificar rutinas de explode y bridge para que desalojen la tarea

Básicamente llaman a la macro desalojar tarea que explicamos en el inciso anterior. Esta macro se agrega en el código de las interrupciones 98 y 108.

7.7. Modo debug

Para implementar el mecanismo tenemos una variable global con 3 estados:0 que es no esta en modo debug, 1 que es esta en modo debug, y 2 que es esta trabado en modo debug (osea que no ejecuta ninguna tarea, esta corriendo la idle). Al apretar la 'y', como tenemos los flags de interrupciones prendidos, cae en una interrupción de teclado. En esta hacemos un call a la función presionar y, que puede pasar de 0 a 1, de 2 a 1, de 1 a 0, aunque nunca de 1 a 2 (si es cero le suma 1, sino le resta). La interrupción de teclado, simplificada, queda:

```
intTeclado:
    pic_finish1()
    teclaApretada<- leerPuerto(0x60 :bufferTeclado)
    si teclaApretada==scancode(y)
        modoDebug= leerModoDebug()
        si modoDebug==2
            recuperarPantalla()
        //presionarY
        si modoDebug ==0
            modoDebug ++
        sino
            modoDebug --
    sino
        //Este printeo lo quitamos para no interferir con el juego
        si esNumero(teclaApretada)
            printearNumero(teclaApretada)
        sino
            si  ¬ esBreak(teclaApretada)
                printearScancode(teclaApretada)
```

Para implementar el modo debug tuvimos que hacer modificaciones en la int de reloj, teclado y excepciones. A continuación haremos un seguimiento desde que se activa el debug hasta que el juego se reanuda, pasando por estas interrupciones.

La primera vez que se aprieta y, cae la interrupción de teclado y se ejecuta presionarY. Esta función hace pasar el modo debug de 0 a 1. Luego, de caer una excepción, en su handler se prende el modo trabado de debug con la función *trabarDebug* que pone el estado en 2. Solo en la excepción se llama esta función que puede pasar al estado 2. En la interrupción de teclado, *presionarY* no puede pasar a 2 (como se ve en el pseudocódigo de arriba). En la excepción, cuando se pasa a modo 2, también se guarda la pantalla del juego (en un arreglo en el kernel, *copia_pantalla*), antes de imprimir el panel del debug y después de matar al lemming en pantalla. Este orden es para que la pantalla quede actualizada con la ultima muerte, y que no se pinte un casillero una vez que ya estamos en modo debug.

Para matar el lemming en pantalla usamos *autodestruirEnPantalla*.

Para explicar esta función, hace falta saber donde esta cada lemming en la pantalla. Para esto tenemos un arreglo de coordenadas, que en caso de tener un numero distinto de cero en longevidad, tiene las coordenadas del lemming asociado al índice. Si no esta vivo el lemming (longevidad =0) la entrada en el arreglo coordenadas es invalida.

Ejemplo:

$$longevidad = [0, 3, 6, 0, 4...]$$

$$coordenadas = [[20, 12], [3, 0], [20, 12], [20, 33], [10, 20]...]$$

Con esta configuración, el primer lemming de amalin esta muerto, por lo tanto la coordenada en índice 0 es invalida. Sin embargo, el lemming con índice 2, el segundo de amalin, esta vivo y esa coordenada es valida.

autodestruirEnPantalla primero consigue el numero de tarea con la función que explicamos antes, trAnroTarea. Con este nro indexa el arreglo de coordenadas y obtiene la coordenada donde esta el lemming a destruir. Va a la pantalla en esa coordenada y reemplaza el lemming por lo que había en el mapa de nivel inferior, osea casillero libre, escombro o puente, que se explica en el ej 8.

Volviendo a las otras cosas que hace la excepción, para printear la pantalla del modo debug se utiliza una función que recibe por pila casi todos los valores que printea:

- Los registros de propósito general, directamente los pusheamos.
- La pila. Dentro de la función de *imprimirDebug* accedemos, de estar mapeado, a lo que hay guardado en los últimos 3 dwords pusheados (menos si no hay 3).
- Para los registros de control, los movemos primero a registros y luego los pusheamos.
- El eflags, el eip, el cs, error code, ss y esp salen de la pila que pusheo la excepción (el que tenia la tarea antes). Para el error code, lo primero es chequear si la excepción tiene uno con la función *hayEc*, que lo hace según el manual. Si fuese una excepción sin error code, no habría nada pusheado. Si ocurriera lo que queremos hacer es agregar el errCode en donde estaba esp-4 justo al caer la interrupción y trasladar todos los registros pusheados adelante de eso. Para lograr este resultado popeamos todos los registros que pusheamos al comienzo, pusheamos un error code dummy y volvemos a pushear esos registros, como si la interrupción si hubiese tenido un error code (este mecanismo se puede apreciar en el pseudocódigo del handler de excepciones que se ve en la sección 8.6).
- El esp del contexto actual en donde se llama la función es pasado como parámetro, pero como no se requiere el esp0 de la tarea, no se printea en el debug.

Una vez que se pasaron todos los inputs de *imprimirDebug* por pila se pasa a la función. Acá se realiza la impresión de todos los parámetros. En particular, para la pila de la tarea, esta puede no tener pusheada 3 valores. Para esto tenemos un if que va preguntando cuan abajo numéricamente esta el esp de la tarea respecto de su esp inicial. Printeamos tantos registros como dwords avanzados este el esp respecto de este (avanza para abajo numéricamente). Si no hay nada printeamos un valor dummy.

Ni bien imprimimos el modo debug desalojamos la tarea. Ahora, como el debug esta en 2, la interrupción de reloj va a leer el estado del debug y como esta en 2 va a saltar a iret directamente. Entonces se queda loopeando en la idle, a la espera de una interrupción de teclado. La interrupción de reloj modificada se puede ver en el pseudocódigo de la sección 8.7.

Cuando cae nuevamente una interrupción de teclado con la 'y', el debug pasa de modo 2 a 1 con *presionarY*, y solo en este pasaje de modo, la interrupción llama a la función *recuperarPantalla*, que recupera la pantalla que se guardo cuando cayo la excepción. Esto se ve en el pseudocódigo de arriba.

Ahora, estamos nuevamente a la espera de otra excepción para trabar el debug (modo 1). Si apretamos 'y' denuevo antes de cualquier excepción, volvemos al modo 0. En este modo la interrupción de reloj funciona normalmente al igual que en modo 1. La diferencia es que en modo 0 las excepciones solo desalojan la tarea, no traban el juego con el debug ni muestran una pantalla de debug.

8. Ejercicio 8

8.1. Iniciar la pantalla del juego

Para este ítem cargamos los obstáculos en el mapa, que estaba verde (de ejercicios anteriores). Para esto, en vez de printear espacios verdes en el buffer de video, recorrimos el mapa brindado por la cátedra

(orga2 con corazones): printeamos cada carácter y le pusimos su color correspondiente(*printearObstaculosIniciales*).

8.2. Implementar interrupción move y verificar que no avance cuando hay obstáculos/paredes

Desde la interrupción 88, sabemos que en eax nos llega la dirección en la que hay que moverse, entonces lo pasamos a una función en C que se encarga de resolver el movimiento. Primero la función obtiene la posición a la que se mueve el lemming a partir de la dirección movimiento y la coordenada que figuraba para ese lemming en el arreglo de coordenadas (modificando las coordenadas originales). Con la posición a la que se mueve, si no esta en rango, restaura la coordenada y devuelve el numero que indica que se intento mover fuera de rango. Si esta en rango, la función lee la memoria de video. En base a si esto es pared, borde, lago o mapa libre el lemming se mueve o no. Si se mueve, volvemos a poner el casillero en donde estaba con lo que tenga el mapa de nivel inferior y el casillero al que se mueve con el lemming correspondiente, beta o ama (lo del mapa de nivel inferior se explica en el siguiente inciso). Si no se mueve, volvemos a poner las coordenadas de ese lemming como estaban (es decir sin moverse, las originales) y devolvemos el numero que indica porque no se pudo efectuar el movimiento.

Finalmente, cuando la función en C retorna a la interrupción, pasamos el eax que devuelve esta función a la cierta ubicación en la pila. Esta ubicación es la del eax que se pusheo en el pushad al comienzo de la interrupción. Asi, cuando la interrupción ejecute el popad queda el resultado del move en eax.

Move también cuenta con una función que revisa si se gano en cada movimiento. Esto se explica mas adelante, en la lógica de finalización del juego.

8.3. Implementar explode y verificar que exploten paredes y mate Lemmings alrededor

En el servicio de explode llamamos a una función de C que se encarga de destruir paredes lindantes, matar lemmings y autodestruir al lemming que ejecuto el explode. Para las paredes explotadas, se pinta la pantalla con una cruz. Para los lemmings, se pinta la pantalla con un carácter del mapa de nivel inferior (y su color correspondiente). Este mapa tiene puentes o escombros, o una casilla libre si es que inicialmente había una en donde estaba el lemming. Los escombros de este mapa al igual que los puentes quedan guardados si alguna vez un lemming murió para crear uno.

La función de C, *matarAlrededores*, pasa por cada posición a distancia 1 manhattan y verifica que hayan lemmings o paredes. Si hay lemmings los mata. Para detectar los lemmings, vemos en el mapa que esta en el buffer de video si hay una 'A' o una 'B' en el carácter del casillero. Para poder matarlos en longevidad, necesitamos el nroTarea pero tenemos la coordenada. Introducimos la función que dada una coordenada encuentra el nroTarea (*buscarLemmingDeCoordenada*). Esta función recorre las coordenadas de lemmings vivos y cuando encuentra la coordenada del lemming lindante devuelve su nroTarea. Para matarlos les pone 0 a su correspondiente entrada en el arreglo longevidad. Con esto no son ejecutados mas por el scheduler (*matarLemmingDeLongevidadYDesmapear*). Luego se los reemplaza en la pantalla por lo que había antes (*reemplazarXCasilleroDelMapaInferior*).

A continuación explicamos estas dos funciones

matarLemmingDeLongevidadYDesmapear:

Recibe el nroTarea del lemming que va a matar poniéndole 0 en longevidad, y desmapea todas las direcciones virtuales que podría tener mapeadas (de $0x400000$ a $0x1400000$). Esto se hace con *mmu_unmap_page*, que de no estar mapeada la pagina no hace nada.

reemplazarXCasilleroDelMapaInferior:

Como precondicion, sabemos que la posición recibida (de la que se va el lemming) no es una pared ni un lago , así que nunca printeamos un lago ni una pared. Y tampoco es una posición fuera de juego.

```
rempPorMapaInferior(f,c)
    Si (f,c) en mapaNivelInferior es
        puente => printeo 'X'
        escombro => printeo '+'
        casilleroLIbre=> printeo '.'
```

Finalmente, el lemming que explota se autodestruye (*autodestruirseEnPantalla*). Esta desaparición del lemming también considera el casillero que había antes. Al volver al servicio, se desaloja la tarea: como vimos anteriormente, se lo quita del arreglo longevidad (setea en 0) y se salta a la tarea idle para esperar al próximo tick de reloj.

Por otro lado, **el move se vio influenciado** por el explode, y posteriormente también por el bridge. Estos dejan un escombro o puente, y si otro lemming pasa por ahí, deben dejar el escombro o el puente como estaba. Es decir, debemos tener constancia de que se dejo un escombro o un puente, para que al irse el lemming de ahí, el puente/escombro siga en esa posición.

Para esto creamos el **mapa de nivel inferior** mencionado anteriormente. El mapa inicialmente es el del juego (lo cargamos antes de habilitar interrupciones, en kernel.asm) pero se va modificando con los escombros/puentes que se van creando. Así, al restaurar una posición por la que paso un lemming, en vez de llenar con una casilla libre llenamos con lo que hay en en este mapa, con *rempPorMapaInferior*.

8.4. Implementar el servicio bridge y verificar que funcione

Como sabemos, este servicio esta asociado a la int 108. Desde la interrupcion, llamamos a la función bridge de C.

Bridge empieza borrando el lemming de la pantalla, porque sabemos que aunque intente crear un puente en un casillero donde no hay un lago muere igual. Luego se obtiene la posición del puente tentativo a partir de la posición en la que esta el lemming que llamo al servicio y la dirección recibida. Al igual que con el move, modificamos las coordenadas del lemming, solo que en este caso no nos interesa restaurarlas porque muere. Después nos aseguramos que la posición del puente tentativo este dentro del tablero de juego. Y si esta, creamos el puente en la pantalla y lo guardamos en el historial de escombros/puentes, el mapa de nivel inferior. Finalmente en la interrupción se desaloja la tarea con la macro ya explicada.

8.5. Implementar la lógica de creación de lemmings

Para crear lemmings en la pantalla usamos la función *nacerEnPantalla* que recibe el nroTarea del lemming que nace. Antes de hacerlos nacer, determinamos si ya hay un lemming en la posición inicial. Para verificar esto buscamos en las coordenadas si hay uno vivo que tenga una posición inicial, tanto para amalin como para betarote. Si hay, no creamos uno nuevo. Si no hay, creamos uno nuevo, lo printeamos en pantalla y guardamos su coordenada inicial en el arreglo de coordenadas. Una vez en esta función les asignamos la coordenada inicial de donde nacen los lemmings del equipo correspondiente en el arreglo de coordenadas. Este arreglo de coordenadas contiene las coordenadas de los lemmings, que de estar vivos, son validas (la coordenada en el índice i corresponde a la tarea i-ecima, enumerándolos como definimos anteriormente). La función *nacerEnPantalla* devuelve 1 si logro nacer, y 0 si no.

Esta función se llama en *nacerSiguiente*, que a su vez esta otra es llamada por el scheduler cada 401 ciclos dentro de la función *chequearCiclos*. Si efectivamente habia lugar en la pantalla para que un lemming nazca en la posición inicial, *nacerEnPantalla* devuelve un 1 (nació es 1) y se le da a ese lemming el siguiente valor de longevidad para su equipo. Es decir, busca el máximo elemento del arreglo longevidad considerando solo las posiciones de la paridad del equipo, y le suma uno. Osea, en longevidad pasa de ser cero a ser el de máxima longevidad (numericamente) de su equipo. También se actualiza el panel (le suma uno al contador de lemmings de ese equipo y lo printea en pantalla con *actualizarPanelSpawneados*).

Algo a tener en cuenta es que cada vez que nace un lemming, **refresheamos su TSS** de manera muy similar a *tss_task_init*. Primero preservamos las variables del contexto actual: esp0 y cr3. Una particularidad es que como no conocemos el esp0 con que inicializo la tarea, lo conseguimos a partir del esp0 que quedo en la tarea que muere. Si la pila se había usado, le limpiamos los primeros 3 nibles y le sumamos una pagina, y si no la dejamos como esta. El cr3 es el mismo que se cargo cuando se hizo el primer mapeo, no necesitamos cambiarlo, pero si preservarlo.

Una vez que guardamos dos registros, pisamos la tss del lemming que nace (en el arreglo de TSSs) con un tss limpio. Y a este nuevo tss le cargamos el esp0 y cr3 que preservamos. De esta manera reutilizamos las TSSs, por lo que solo necesitamos 10 TSSs.

Para los llamados de inicialización de lemmings, el segundo input de la función es siempre -2. Esto es un rebusque para que siempre refreshee la tss del lemming que va a nacer en los 401 ciclos, es decir que no postergue este evento. Pero porque postergamos el refresheo en ciertos casos?

Decidimos **postergar el refresheo de la TSS** en ciertos casos porque nuestra función *nacerSiguiente* también se encarga de hacer revivir al lemming mas longevo cada 2001 ciclos. Y cuando revive un lemming por ser el mas longevo, puede ocurrir que al hacerlo nacer se este ejecutando ese lemming. Esto seria un problema, porque cuando salta fuera de esa tarea se pisaría lo que se quizo limpiar (la TSS) con los registros que ya habían en el procesador. Si caemos en el caso que el lemming a matar y revivir era el que se estaba ejecutando, podemos asegurar que el siguiente lemming a ejecutar no va a ser el que revivió. Esto es porque están todos vivos, y siempre va a pasar por otro antes de pasar por el lemming que postergo su refresheo. Entonces, cuando sucede esto, decidimos hacer el refresheo en el ciclo siguiente.

Para postergar el refresheo nos guardamos en una variable el nro de Tarea del lemming del que hay que refreshear su TSS. Esta variable al comienzo es -1, que indica que no se postergo ningún refresheo. En chequear ciclos revisamos si esta tiene un numero de tarea o si es -1. Si tiene un numero de tarea le asignamos un -1 y refresheamos la TSS de esa tarea. Sino no se hace nada.

Un **caso borde** es por ejemplo, si muere el mas longevo desde una posición del mapa y trata de revivir en la posición de inicio de su equipo, pero ya hay un lemming de su equipo allí (de otro equipo seria imposible, porque ya habría terminado el juego). Como utilizamos la misma funcion para hacerlo nacer *nacerSiguiente*, lo que sucede se encuentra el lemming muerto (el que mato por ser el mas longevo) y se intenta de hacerlo *nacerEnPantalla*. Pero como no puede nacer, devuelve 0, y *nacerSiguiente* termina.

Otro momento en el que nacen lemmings es cuando **muere y revive el mas longevo**. *matarYRevivirViejos* se encarga de esto. Esta función recorre longevidad en búsqueda de 2 cosas por equipo: el mas longevo (el mínimo numéricamente), y un b̈oolq̈ue indica si están todos vivos. Con esta información, si están todos vivos matamos al mas longevo (lo sacamos de la pantalla, lo ponemos en cero en longevidad y le desmapeamos las paginas de memoria compartida) y lo hacemos renacer (usamos *nacerSiguiente* pero con el nroTarea del que queremos hacer revivir). Cuando *nacerSiguiente* recibe un nroTarea valido, posterga el refresh de la TSS si esta tarea se esta ejecutando. Además, como estaban todos vivos y matamos uno, *nacerSiguiente* hace nacer al que matamos.

8.6. Fallos de pagina y memoria compartida bajo demanda

Para compartir memoria bajo demanda entre los lemmings, nos topamos con un problema. A medida que los lemmings iban mapeando nuevas paginas de memoria desde las direcciones lineales $[0x400000; 0x1400000)$, necesitabamos que un lemming que **trataba de acceder a una dirección lineal que ya estaba mapeada por otro lemming de su equipo** lo haga a la misma dirección física que lo mapeo el lemming que requirió esa pagina. De lo contrario, no le estaríamos dando acceso a la misma memoria, y no seria c̈ompartida.

Para tener **constancia de que mapeos hizo cada equipo**, guardamos dos arreglos (uno para cada equipo) de 4096 direcciones físicas, empezando todas en cero. Como estos arreglos tienen un tamaño importante, decidimos guardarlos en el área libre del kernel en tiempo de ejecución. Esto se hace en la funcion *init_estructuras_memoria_compratida*. Básicamente pide next free kernel page para el arreglo de amelin, suma 3 paginas y pide denuevo next free kernel page para betarote y suma otras 3 paginas. Con esto cada puntero tiene por delante 4 paginas para indexar 4096 dwords. Una vez que creamos los punteros, tomamos el que viene en la dirección mas baja y llenamos 8 paginas con ceros. Esta función la llamamos en el kernel antes de habilitar las interrupciones.

En los arreglos de direcciones físicas, de estar alguna **en cero, significa que la pagina correspondiente al índice de esa dirección no esta mapeada**. Los índices representan el numero de las paginas que mapean direcciones lineales van en el rango $[0x400000; 0x1400000)$, es decir un total de 4096 paginas, contando desde la que va de $0x400000$ a $0x400fff$ con índice cero hasta la que va desde $0x13ff000$ a $0x13ffffff$ con el indice 4095. De tener una dirección física valida en algún índice de algún arreglo significa que la pagina que le corresponde al índice fue mapeada por un lemming de ese equipo (según que arreglo es, amalin o betarote). Una dirección física valida podría ser alguna entre $0x40a000$ y $0x240a000$, ya que podríamos mapear 16MiB para cada equipo.

Por ejemplo, al realizar un page fault en una dirección lineal de memoria compartida, sabemos que esa excepción es arreglable. Entonces al manejar un page fault, caemos en 2 ramas, PF arreglable y PF

no arreglable. El handler de excepciones se modifica de la siguiente manera:

```
excepciones:
    pushad()
    si nroExcepcion==14
        direccionAccedida <- leerCr2()
        si 0x400 000 <= direccionAccedida  < 0x1400 000
            mapearMemoriaCompartida()
            popad()
            iret()
        sino
            modoDebug<- leerModoDebug()
            si modoDebug == 1
                trabarDebug()
                autodestruirLemmingPantalla()
                guardarPantalla()
                si tieneEc(nroExcepcion)
                    popad()
                    pusheamosECDummy()
                    pushad()
                pusheamosRegistrosControl()
                imprimirDebug()
    desalojarTarea()
```

En la excepción, cuando el **pf es arreglable**, como vimos arriba, se hace un call a memoria compartida. Acá revisamos en el arreglo de paginas mapeadas del equipo que hizo el acceso invalido si en el índice que corresponde a esa dirección accedida hay un cero o una dirección valida. Notemos que el cero no es una dirección física valida para memoria compartida. Si hay una **dirección valida**, creamos un mapeo desde el cr3 actual con la dirección base del marco de pagina que engloba a la dirección que se quizo acceder hacia esa dirección física que figura en el arreglo. Si esa **dirección en el arreglo es cero**, mapeamos una nueva pagina desde la misma dirección lineal hacia una nueva pagina libre de usuario, que se encontraría por arriba de la dirección física 0x40a 000 (las diez primeras paginas de usuario tienen las pilas de nivel 3 de las 10 tareas). Luego de hacer esto, en el arreglo correspondiente al equipo indexado con el índice de la dirección lineal, se guarda la dirección física que se mapeo, para que otros lemming accedan a los mismos datos después de mapear esa dirección lineal a la misma física.

Y si el **pf es no arreglable, se desaloja la Tarea**. Otra cosa de la que nos aseguramos es que cada vez que mapeamos una nueva pagina llenamos todo el marco de pagina con ceros. Fue un requisito de un test.

8.7. Lógica de finalización del juego

Creamos un chequeo extra en la interrupción de reloj que revisa si la variable *terminoElJuego* esta en 1. Si esta en 1, la interrupción de reloj no va a hacer nada, simbolizando que termino el juego. Con todas las modificaciones ya hechas, la interrupción de reloj queda de la siguiente manera:

```
interrupcionReloj:
    picFinish1()
    printClockSistema()
    si terminoJuego==0 && modoDebug!=2
        siguienteTarea<- sched_next_task
        printClockTarea()
        tareaActual<- str()
        si tareaActual!=siguienteTarea
            saltar(siguienteTarea)
    iret
```

Para setear correctamente la variable *terminoElJuego*, debemos registrar si cada move fue hacia una posición victoriosa o no. Esto se ve en la columna de la coordenada hacia la que se mueve el lemming (de estar en rango). Si por ejemplo un amalin se mueve y llega a una posición victoriosa la columna es la que esta mas a la derecha de la pantalla(offset 79 en la pantalla). Y si un betarote se mueve a una posición victoriosa la columna es la que esta mas a la izquierda de la pantalla (offset 0). Este chequeo se hace con la función hasWon, constantemente, por cada movimiento de cada lemming. Si termina el juego, como la interrupción de reloj no hace nada, el scheduler no va a intercambiar entre tareas y va a quedar en loop ejecutando la idle a la que salto el ultimo move. Se imprime el equipo ganador arriba de la pantalla.

9. Conclusión

A lo largo de este trabajo práctico fuimos repasando los conceptos de la programación de sistemas operativos de forma secuencial, construyendo ,ejercicio a ejercicio, desde un sistema básico hasta un sistema capaz de correr tareas concurrentemente, manejar interrupciones y ejecutar con dos niveles de protección.

10. APÉNDICE

10.1. Paginación

Como se ve en la figura 1, el campo directory se utiliza como offset para indexar el page directory y crear un page directory entry (si fuera necesario), el campo de table se usa como offset para indexar el page table y crear un page table entry, y el campo offset sumado a la base a la que apunta el PTE se usa para acceder a la memoria física que se quiere mapear.

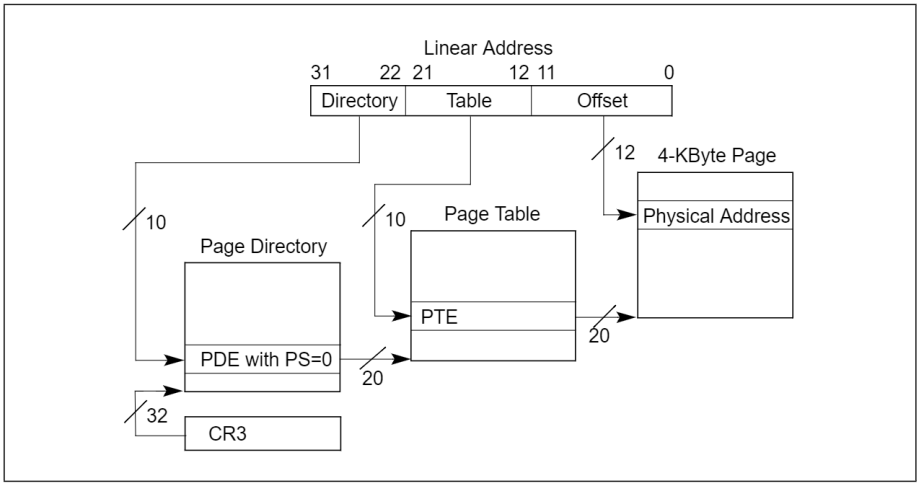

Figura 1: Esquema de traducción de una dirección lineal a una página de 4KB con paginación de 32 bit.