DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Programación de Sistemas Operativos

Organización del Computador II

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Matias Cozzani	915/19	matcozzani@gmail.com
Marco Sanchez Sorondo	708/19	msorondo@live.com.ar
Joaquin Gonzalez Vandam	720/19	joaking2011@gmail.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, motivación y objetivos

Los procesadores de la arquitectura Intel x86 proveen dentro de sus funcionalidades, un framework para el desarrollo de sistemas operativos. Queremos implementar uno muy elemental que sea capaz de inicializarse, captar señales teclado de manera tal que un usuario pueda jugar a una competencia 'Lemmings' de manera segura (es decir, con una interfaz que proteja la información interna del sistema). Para ello nos proponemos programar un kernel y las estructuras necesarias para implementar el sistema, explotando las funcionalidades proveidas por Intel.

2. Implementación

2.1. Ejercicio 1: Segmentación

2.1.1. a.

Se define a la GDT como un arreglo de elementos de tipo 'gdt_entry_t' (que ocupan 8bytes cada uno). El primer elemento de este arreglo fue seteado todo en cero tal como lo exige el manual. Para los cuatro segmentos pedidos en el enunciado se establece como base 0x000000 y límite 0x330FF (aproximadamente 817MiB) (pues el sistema de segmentacion debe ser 'flat'). Para ambos segmentos de nivel 0 se setea el flag DPL en 0 y para ambos segmentos de nivel 3 setea el flag de DPL en 3. Además, para los segmentos de datos se les pone el 'type' en READ/WRITE y los de código en EXECUTE/READ. Como la el espacio para estos segmentos es mayor a 1MiB se setea la granularidad en 1 para todos. El atributo 's' se setea en 1 para los 4 pues se trata de segmentos de codigo/datos. Por último, en todos se setea el bit de presente en 1 pues estamos inicianizando las entradas. Debe notarse además que las entradas de los descriptores de segmento comienzan en el indice 8 del arreglo de entradas de la GDT.

2.1.2. b.

Como primera instruccion en nuestro kernel, deshabilitamos las interrupciones con la instruccion CLI, principalmente para poder inicializar las estructuras del sistema sin ser interrumpidos. Se carga el registro GDTR con la direccion que se estableció como base de nuestra GDT con la instrucción LGDT [GDT_DESC]. Se activa el bit del CR0 que indica si el sistema en modo protegido y luego se salta al codigo del modo protegido del sistema con un jump far. Para ello se indica el offset para el descriptor del segmento de código de nivel cero en la GDT (es 64, porque entre el descriptor nulo y el inicio de los descriptores dejamos 7 entradas sin llenar, tal como se dijo en 2.1.1.).
Una vez en modo protegido se setean los selectores de segmentos con el valor del offset dentro de la GDT para el descriptor de segmento de datos de nivel cero. Además, se setea la base de la pila (ebp) en 0x25000 con la etiqueta ESP_INIT.

2.1.3. c.

Se define otro elemento en la GDT para el segmento de video, con base en 0xB8000 como lo detalla el gráfico del enunciado. Como esto es un segmento para almacenar los datos (tipo=2) de video, establecemos como límite el espacio de memoria ocupado para describir a cada pixel (2 bytes por pixel, pantalla de 80x40 nos da 8000 bytes) menos 1 (en hexadecimal, 0x1F3F). Como ésto es menor a 1MiB entonces la granularidad será de a byte (0). Como será ejecutado por el kernel, se setea dpl en 0.

2.1.4. d.

Para inicializar la pantalla primero se cicla a lo largo de sus 80x40 pixeles superiores (con base en el inicio del segmento de video), pintando su fondo y frente de color azul y sin ningún caracter (0x1100). Al llegar al primer pixel de los 80x10 restantes, se salta a la esquina superior izquierda de las 'cajas' de cada equipo y se pintan ambas en cada iteracion completando de a filas de 10 pixeles.

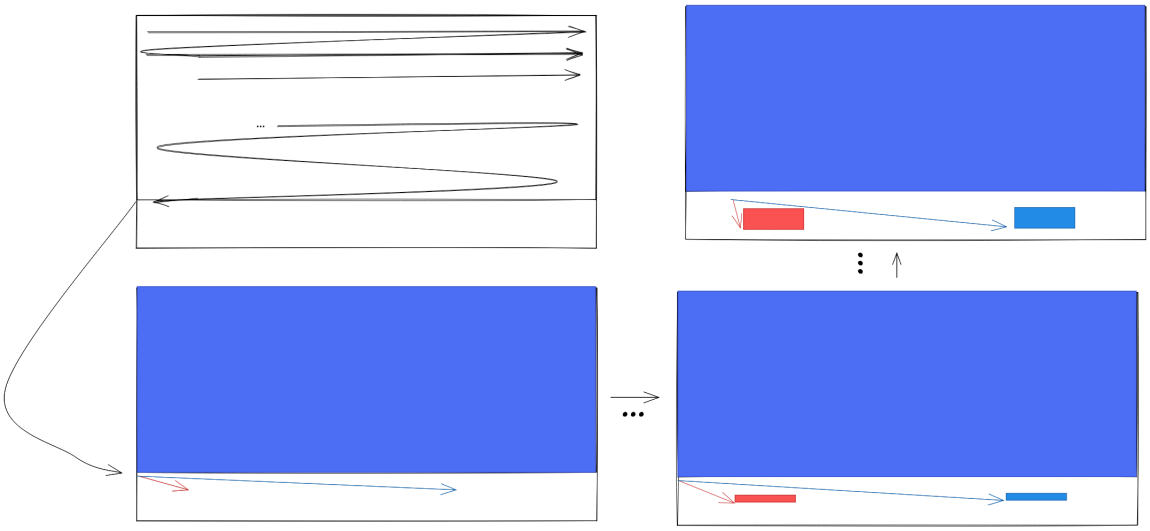

Figura 1: Llenado de pantalla

2.2. Ejercicio 2: Descriptores de interrupción

En *idt.c* definimos un inicializador estándar de descriptores de interrupciones que completa el descriptor de nuestras interrupciones, cuando necesitamos un descriptor distinto de los estándar lo modificamos en la función IDT_INIT. El inicializador define como offset una referencia al inicio de la interrupción cuyo número se pasó como parámetro, en el segmento de código. Setea el bit de presente en 1, el DPL en 0. Luego, para las interrupciones 88, 98 y 108 el DPL se setea en 1 para que se ejecuten a nivel usuario (son los servicios para jugar).
Para las excepciones del procesador, definimos en *isr.asm* una función estándar que: 1. Preserva todos los registros de propósito general, llama a la función MODODEBUG que chequea si se oprimió la tecla 'y' para ingresar al modo debug, llama a la función DESALOJAR (que desactiva la tarea actual del scheduler) y luego se hace un jmp far a la idle.
Para que el procesador utilice la IDT, desde el kernel se llama a la función que la inicializa, cargamos la direccion de la IDT en el registro IDTR, configuramos el PIC y habilitamos las interrupciones.

2.3. Ejercicio 3: Inicialización de la IDT e implementación de interrupciones

2.3.1. a. y b.

Se llama nuevamente al inicializador de descriptores de la IDT para las interrupciones de teclado (33), de reloj (32) y de los servicios (88, 98, 108).
La rutina asociada a la interrupción de reloj llama a PIC_FINISH, que avisa al IC que la interrupcion fué atendida, luego llama a SCHED_NEXT_TASK para saber si hacer un jmp far hacia la siguiente (en caso de haber mas de una tarea activa en el scheduler) o si continuar con la ejecucion de la actual (llamando a next_Clock como lo exige el enunciado).

2.4. Ejercicios 4 y 5: MMU

2.4.1. a. y b.

Inicializamos dos punteros: uno para la tabla de paginas y otro para la tabla de directorios (ambos en las direcciones indicadas por el enunciado), completamos las 1024 entradas de cada una de las tablas con ceros. Establecemos como base de la primera entrada de la tabla de directorios a la base de la tabla de páginas. En cuanto a los atributos, seteamos el bit de presente en 1 para indicar que la entrada está inicializada, el de read/write en 1 y el de u/s en supervisor de manera que las tareas de nivel usuario no tengan acceso al kernel. Luego se inicializan todas las entradas de la tabla de paginas con los mismos

atributos. Para que haya identity mapping, seteamos la dirección física de cada entrada en la tabla de paginas con el indice que tienen en la tabla de paginas. Desde el kernel, luego de inicializar la mmu general (descripta en el próximo ejercicio), se ejecuta la instrucción CALL MMU_INIT_KERNEL_DIR.

2.4.2. (Ejercicio 5) a. y b.

En MMU_INIT se inicializan dos variables: el inicio de las paginas libres del kernel y el de las páginas libres del usuario.
La función para mapear páginas realiza el mapeo entre una direccion fisica y una virtual, recibiendo además el contexto para el cual se realiza el mapeo y los atributos deseados. Levanta los índices dentro de la tabla de directorios y la tabla de paginas que se encuentran en la dirección virtual. Si la entrada de la tabla de directorios no tiene asociada una tabla de paginas, entonces se crea una nueva tabla de paginas nula, se setea el bit de presente y se asocia su direccion a la entrada en la tabla de páginas. Si el bit de presente sí estaba en 1 entonces la tabla de páginas asociada existe y por lo tanto el paso anterior se omite. Luego (para ambos casos) se escribe en la entrada de la tabla de páginas -cuyo índice levantamos al principio de la función- y se escribe en la dirección física la dirección física pasada por parámetro y se setea su bit de presente en 1. Para poner los atributos de la nueva entrada lo que hacemos es la operacion logica .º.entro los atributos que ya tiene la PDT y la PT con los recibidos por parametros, de esta forma si ahora requerimos algun atributo prendido que antes no lo estuviera, lo podemos obtener mantienendo los demás. Al final de la función se ejecuta la función tlbflush() que se encarga de invalidar todas las entradas de la TLB.
Para el desmapeo, simplemente se deshabilita el bit de presente en la entrada de la page table y se extrae la dirección física que almacenaba para devolverla.

2.4.3. c.

La función de mapeo de tareas se encarga, primero, de crear una tabla de directorios para la tarea, piendo una pagina del area libre del kernel. Luego mapea el kernel con identity mapping, valiendose de la función map_page ya defina anteriormente. Finalmente mapea, utilzando nuevamente map page, las áreas requeridas para la tarea. Primero mapeamos dos paginas para el código de la tarea a partir de la dirección virtual 0x08000000 como pide el enunciado: para hacerlo es necesario saber si esa dirección virtual debe mapearse para la dirección de código que se corresponde con el equipo A o el B, para resolver esta situación la función recibe por parámetro la dirección fisica a la cual debe atar la dirección virtual dada, en modo usuario para solo lectura. Por último mapeamos una última página a partir de las dirección 0x08002000, nuevamente en modo usuario pero para lectura escritura ya que está será la página designada para la pila de la tarea.

2.5. Ejercicio 6: TSS

2.5.1. a.

Inicializamos las entradas de la GDT que describen a la TSS utilizada por la tarea inicial, la tarea idle y aquellas que serán utilizadas para las tareas de los lemmings. Al tratarse TSSs, establecemos su límite en 67 (lo cual desde la base seria un espacio de 104 bytes, exactamente lo que ocupa una TSS). La base de ambas la establecimos en 0x0 porque las seteamos luego cuando inicializamos la estructuras de las tss a fin de poder hacer arrancar el scheduler. Todas estas tares son de tipo tss de 32 bits. Pondremos su dpl en 0 ya que solo queremos poder saltar a las tareas de nivel 0, no queremos que las tareas sean capaces de saltar entre ellas. Además seteamos el bit de presente.

2.5.2. b.

(Definido en tss.c) Utilizamos la misma pila que el kernel (a partir de 0x25000) y utilizamos su mismo cr3 como lo solicita el enunciado. Se setea como segmento de código el de código de nivel cero y para el resto se setea el de datos de nivel cero. Esto se debe a que queremos guardar el código y datos de ésta tarea en nivel cero para ejecutarla desde nivel cero. El eflags se setea en 0x202 para permitir que la IDLE

sea interrumpida. El instruction pointer de IDLE lo seteamos con la dirección 0x0001C000 como lo exige el enunciado.

2.5.3. c. y d.

Seteamos la base de la entrada de la GDT correspondiente a la tarea inicial con un puntero a la estructura tss_initial inicializada en el archivo tss.c y hacemos lo mismo para el discriptor de la tarea idle. Para estos partimos con shifts los punteros y los guardamos en los componentes que corresponden.

2.5.4. e.

Para esto utilizamos las siguientes instrucciones:

```
mov ax, TSS_INITIAL; cargamos en ax el puntero a la tss de la tarea inicial
ltr ax ; guardamos en el registro TR el puntero a la TSS de la tarea inicial para poder
;luego recuperar el contexto

jmp TSS_IDLE:0 ; saltamos a la tarea IDLE.
```

2.5.5. f.

Esta función crea una TSS como paso anterior a la creación de una tarea. Para ello, primero inicializamos un arreglo de elementos tipo tss, todos nulos, de tamaño 10, donde guardaremos la información de las tss de las tareas y tenerlas de guardas en memoria de manera global. Una vez dentro de TSS_CREATE, llamamos a mmu_init_task_dir para realizar el identity mapping del kernel y para mapear el sector de código y el stack de la tarea, recibiendo la dirección fisica donde se ubica la tarea. Luego además pedimos una nueva página del área libre del kernel para la pila de nivel 0 de la tarea.

Inicializamos los selectores de segmento de codigo y de datos con el offset a los segmentos de codigo y de datos de nivel 3, le asignamos a cr3 el valor del PDT obtenido anteriormente con la funcion init_tas_dir. Seteamos el registro EFLAGS en 0x202 para habilitar interrumpir la tarea. Luego asignamos a EIP la dirección virtual de código y al ESP la dirección virtual del stack. A cada uno de los selectores de segmento se realiza un bitwise or con 0x3 para setear el RPL en 2. Para iomap establecemos el valor 0x68, que es uno mas que el límite de la TSS... Esto se debe a que para no realizar mapeo E/S (según el manual) hay que asignarle a ésta variable un valor mayor o igual al tamaño de una TSS (0x68). Luego seteamos el ss0 poniendole el offset en la gdt del selector de segmento de datos de nivel 0 y al esp0 la dirección que obtuvimos al pedir una pagina libre del área del kernel y le sumamos el tamaño de un página(4kb) para que quedo al fondo de la misma Finalmente, completamos en la gdt, en el indice que se corresponde con el indice en el arreglo de tss de la tarea creada más 15 ya que los descriptores de tss de las tareas están a partir ese indice, la base del descriptor con una referencia a la tss creada en el array.

2.6. Ejercicio 7: Scheduler

2.6.1. a.

Definimos en sched.h nuestra estructura para scheduler y nuestra estructura para cada tarea. Los atributos del scheduler son modoDebug (indica si el modo debug esta activado) (bool),debugging (indica si esta abierta la ventana del debugger) (bool), tasks[10] (arreglo de las tareas), currentTask (índice de la tarea que se está ejecutando), prevTask (índice de la tarea anterior a la actual). // Para la estructura task se ponen los atributos isActive (para ver si el scheduler la incluye en la ronda de ejecucion), el indice del descriptor en la GDT de la tarea con taskIdx
La funcion sched_init en sched.c inicializa el scheduler con modoDebug desactivado, debugging desactivado, currentTask y prevTask en 10 (la idea es que vaya de 0 a 9 para representar los indices de cada tarea pero con 10 denotamos que la tarea actual es la IDLE, que es el caso a la hora de incicializar),

nextTask en 10 por la misma razón (mientras no se realicen interrupciones) y finalmente se incicializa cada tarea con isActive en false.

2.6.2. b.

SCHED_NEXT_TASK La idea es buscar la próxima tarea a partir de saber cual es la tarea anterior. Pero estos genera problemas en la alternacia de las tareas de cada equipo cuando la tarea anterior y la actual son del mismo equipo, lo cual sucede cuando un equipo se queda sin lemmings mientras el todavía tiene tareas activas. Esto es por que tenemos como precondición al buscar la tarea que sigue que la tarea anterior a la actual sea del otro equipo. Como la tarea actual es el índice en arreglo de tareas y los Amalin tiene tareas de índice par y los Betarote impares, cuando la paridad de la tarea actual y la previa son iguales entonces cuando agarremos la tarea previa estamos buscando tareas del mismo equipo cuando antes deberia chequear si tenemos alguna disponible del equipo contrario para cambiar. Para solucionar esto creamos este ternario. Si la tarea anterior y la previa tienen la misma paridad, entonces vamos a querer que aux sea un índice de paridad contraria, entonces sumamos un numero impar a la tarea previa lo que soluciona el problema.

De esta manera podemos utilizar nuestra función sched_next_task_aux tranquilos. Esta función recibe el indice a partir del cual debe buscar la proxima tarea, cuyo indice sea de la misma paridad que el del recibio, que esté activa y devolver el indice de su descriptor en la gdt. Entonces primero buscamos a partir del indice de la tarea anterior sumado en dos, ya que no queremos volver a revisar la anterior que sabemos que ya se ejecutó. De todas formas si no se encuentra con la auxiliar, desde el indice pasado por parametro en adelante una tarea activa, se revisa el arreglo completo. Si no encontramos tarea activa entonces devolvemos el indice en la gdt de la tarea idle.

Cuando no se encuentra una tarea activa del equipo contrario, es decir la auxiliar devolvió el indice de la tarea idle, se procede a buscar la proxima tarea activa del equipo actual. Si no tomamos a cuenta el caso que la tarea actual y la previa son del mismo equipo(misma paridad), quedariamos buscando siempre para el mismo equipo sin poder volver a saltar al equipo contrario.

Luego se actualiza prevTask y currentTask. Finalmente, si la unica tarea activa era la IDLE se retorna su indice en la GDT y en caso de que no sea la IDLE se retorna el indice en la GDT de la nueva tarea a ejecutar.

2.6.3. c.

En la rutina de atencion de interrupciones de reloj se agrega un 'call sched_next_task' y en caso de que la tarea sea distinta a la anterior se realiza un jmp far hacia su contexto de ejecución.

2.6.4. d.

La rutinas de interrupciones 88 es modificada de tal forma que ejecute como instrucciones finales(antes del iret y el popad) un jumpfar a la tarea idle.

2.6.5. e.

La menera de desalojar la tarea para las interrupciones del procesador es llamar a una funcion killLemming que entre otras cosas llama a la función desalojar, que se encarga de poner para la tarea cuyo indice en el arreglo de tarea del schedule coincide con pasado por parametro, el valor de isActive en falso. Luego saltaran a la tarea idle, para que al caer el proximo tick de reloj se cambie.

2.6.6. f.

Estas tares se desalojan como parte del funcionamiento de la funciones en c que implementa su lógica dentro del juego. Dentro de ellas se utiliza la función killLeming que como ya vimos antes, usa la función desalojar para quitar la tarea del scheduler. Luego realiza el salto a la tarea idle y cambia a la proxima cuando caiga el proximo clock.

2.6.7. g.

En las interrupciones de teclado, se agrega una lectura al buffer de las teclas del PIC para ver si se oprimió la tecla 'y' para activar/desactivar el modo debug con la funcion modoDebugOnOff. Esta función tiene dos tareas, prender el modo debug seteando la componente del scheduler como verdadera si este estaba apagado. Si el mododebug estaba prendido, entonces se debe fijar si tenemos el juego parado con la pantalla del debugger encendida. De ser así apagara la pantalla y pondra la compenente del debuggin del scheduler en falso, que permitira continuar la ejecución del juego.

```
_isr33:
   pushad
   in al, 0x60 ; lectura de la tecla


   cmp eax, 0x15 ; comparacion con el scancode de 'y'
   jne .fin

   call modoDebugOnOff ; cambio del modo en caso de que 'y' fuera oprimida

   .fin:
   call pic_finish1 ; avisarle al PIC que la interrupcion fue atendida
   popad
iret
```

```
void modoDebugOnOff(){
    if(scheduler.modoDebug){
      //Esto va a modificar la variable que use el bucle para cerrar la pantalla
      if(scheduler.debugging == 1){
        scheduler.debugging = 0; //esta es la variable que indica si se esta mostrando o no la pantalla de debug
        setOldScreen();// Se encarga de quitar la pantalla de debugging
      }
    }
    else {
        scheduler.modoDebug = true;
    }
}
```

En las interrupciones de reloj, se chequea si el juego se encuentra en pausa debido a que se esté debuggeando (chequeamos la componente debugging). Luego de la rutina de desalojar las tareas, se llama a getDebugging para cheque la componente del scheduler. En caso de que esté activado se 'esquivamos' todo las otras tareas que realice la interupcion y vamos direco al iret. De esta manera congelamos el juego ya que nunca cambiamos de tarea. En caso contrario dejamos que la interrupción de reloj continue su habitual funcionamiento.

Por último queda ver que se agregó en las interrupciones del procesador para manejar el modo debug. Luego de matar la tarea que produce la excepción, vamos a chequear que el modo debug esté prendido con la funcion getModoDebug que chequea el estado del componente del scheduler. Si no está prendido entonces vamos a saltar directamento a hacer el cambio a la tarea idle por lo que reste del quantum. En caso de que se encuentre activado, se llama a 'setDebugging', que setea en true el atributo debugging del scheduler (que indica si está imprimiendo en pantalla el estado de los registros). Luego, se evalúa si la excepción del procesador tiene un código de error, porque caso en el cual éste exista afecta al juego en la siguiente manera:

- Tenemos que imprimirlo en pantalla.

- Como está en la pila, cambia los offsets lo cual debe ser considerado a la hora de ir a buscar ciertos datos al stack de nivel 0 como el EIP o el puntero a la pila de nivel 3.
- Esto debe ser tenido en cuenta para sacar el error code de la pila antes de realizar el iret.

Dependiendo el caso se procedera de una menera u otra para sacar la info del estado del procesador. Lo primero que haremos será ir a buscar a la pila de nivel 0, el error code si corresponde, el eip que produjo el fallo, los eflags y la dirección del stack de nivel 3. Pusheamos todos estos datos(si no hay error code pusheamo 0xDEAD), pusheamos los selectores segmento, el codigo de excepción y los registros que ya estaban pusheado con el pushad.

De esta manera llamamos a la funcion init_debug_interface que será la encargada de imprimir en pantalla todos estos datos. La función posee la ayuda de algunos switchs para determina escribir el nombre de la excepción en base a su codigo, chequea que si se recibe 0xDEAD como codigo de error se debe escribir guiones, y chequea cuantos elementos podemos imprimir de la pila de nivel 3. Siempre queremos imprimir los ultimos tres datos, pero si la pila esta vacía o tiene menos elementos escribimos guiones en su lugar.

Por último se hace el salto a la idle.

2.7. Ejercicio 8: Gráficos y lógica del juego

2.7.1. h.

El procedimiento en general es el siguiente:

1. Se imprime el mapa en pantalla.
2. Se escriben en pantalla los nombres de los equipos en las posiciones que corresponden.
3. Se imprimen en pantalla los relojes de cada equipo.
4. Se imprimen los puntajes correspondientes
5. Se inicializan los lemmings con lemmings_init() (en las posiciones pares se inicializan los de Amalin, partiendo de la esquina inferior izquierda del mapa y en las impares se inicializan los de Betarote, posicionados inicialmente en esquina superior derecha del mapa. Todos tienen age = 0 al principio).

Para los lemmings vamos a poseer un arreglo de una estructura lemming_t que contendra la información de los lemmings para el juego, como su vida, su edad, su posicion en el mapa y el tipo de terreno sobre el cual están parados.

2.7.2. i.

En la rutina del servicio se utiliza la función auxiliar movLemming que recibe la dirección del movimiento. En movLemming, primeramente se determina cuál es el lemming cuya tarea es la que se está ejecutando (pidiendole al scheduler que nos devuelva la currentTask, ya que los indices de las tareas en el array de tareas del scheduler se corresponden con los indices de los lemmings en el array de lemmings). Luego se determina la dirección a moverse y se verifica que sea posible moverse en tal dirección (no se está en el borde de la dirección en la que queremos movernos, no hay una caja, no hay otro lemming, o en caso de haber un lago que haya un puente). En caso de poder moverse se cambia la propiedad .terrain del lemming(que indica si se encuentra en pasto o puente o pared explotada): esta propiedad nos sirve de ”pivote.ªl momento de ir actualizando el mapa al terreno original que habia antes de que el lemming se moviera y luego se actualiza la posición en el mapa del lemming. En caso de no poder moverse, se inhibe el movimiento y devuelve como resultado de la operación el tipo de bloqueo que no permitió hacer el movimiento.
Finalmente, si fue posible moverse se retorna SUCCESS indicando que fue posible realizar el movimiento. Para mayor nivel de detalle, el código se encuentra comentado.

2.7.3. j. y k.

El servicio boom hace lo siguiente:

1. Se detecta el lemming que va a inmolarse. Se guarda su posicion en variables auxiliares para resolver los efectos de la explosion (romper paredes/matar otros lemmings).
2. Se llama a killLemming, que setea su 'health' en cero, lo borra del mapa y lo desaloja del scheduler.
3. Se chequea una unidad manhattan centrada en la posición del lemming la existencia de otros lemmings o paredes: en caso de que haya pared en una posición, se cambia su caracter por 'x', en caso de que haya un lemming del equipo que sea, se extrae su indice y se lo mata.

En cuanto al servicio bridge, que recibe como parámetro la dirección en la que se construirá. Se procede de la siguiente manera:

1. Se obtiene el lemming que se usará como puente.
2. Se lee la dirección pasada como parámetro para ver si el estado del terreno es efectivamente un lago -L- y que esté dentro de los limites del mapa, caso en el cual se cambia el estado por '+' (simboliza puente) y posteriormente se mata al lemming.
3. En caso de que no haya un lago, no se lleva a cabo ninguna acción.

2.7.4. l.

Para implementar esto crearemos una función game_ticks que será la encargada de, por cada tick de reloj, ver si se requiere crear o matar lemmings. Dentro de la misma, la lógica introducida para la creación de lemmings es:

```
    if(spawn_ticks%401 == 0){
        if(alive(0) < 5){
            lemmingCreate(0);
        }
        if(alive(1)<5){
            lemmingCreate(1);
        }
    }
```

Osea que cada 401 ticks si alguno de los equipos tiene menos de 5 lemmings vivos se crea un lemming nuevo.

```
if(kill_ticks%2001 == 0){

        uint8_t oldLemmingA=0;
        uint8_t oldLemmingB=1;
        for(uint8_t i = 0; i < 10; i+=2){
            if(lemmings[i].age > lemmings[oldLemmingA].age && lemmings[i].health != 0)
                oldLemmingA = i;
            if(lemmings[i+1].age > lemmings[oldLemmingB].age && lemmings[i+1].health != 0)
                oldLemmingB = i+1;

        if(alive(0) == 5){
            killLemming(oldLemmingA);
            lemmingCreate(0);
            return (getCurrentTask() == oldLemmingA || getCurrentTask() ==oldLemmingB);
        }
        if(alive(1)== 5){
            killLemming(oldLemmingB);
            lemmingCreate(1);
            return (getCurrentTask() == oldLemmingA || getCurrentTask() ==oldLemmingB);
        }
    }
```

Osea que cada 2001 ticks se busca el lemming mas viejo de cada equipo para reemplazarlos por lemmings nuevos (con killLemming y lemmingCreate). Hay que tener cuidado con un caso borde en esta sección que es el caso en donde el lemming que muere es el mismo que se está ejecutando, ya que en este caso, al volver realizar el task switch la tss se sobreescribirá con los datos de la tarea corriendo, que es aquella que acabamos de resetear. Para ello entonces vamos a hacer que game_tick nos devuelva un booleano si este caso se dá y en el caso que así suceda en la isr32 antes de hacer el cambio pisar el eip y el esp de la pila de nivel 0 para hacer el retorno al lugar correcto.

La función lemmingCreate recibe como parámetro el equipo del lemming a crear. Dentro de sus lemmings, busca el primero que encuentre 'muerto' y se le restaura la vida, se le posiciona en la posición inicial correspondiente al equipo al que pertenece (si es que no hay un lemming ya en esa posición), se crea una nueva TSS para lemming, se le modifican sus atributos por los de un lemming recién spawneado, se lo imprime en pantalla y finalmente se actualiza la cantidad de lemmings del equipo que lo haya creado.

2.7.5. m.

Para ésto primero se excluye a la excepción 14 de las excepciones definidas de manera genérica con macros y escribimos su código, que hace lo siguiente:

1. Imprime la excepción en pantalla.
2. Se llama a handlerPageFault, que mapea las página que los lemmings solicitan. Sioel mapeo puede realizarse devuelve true y vuelve a ejecutar la tarea que genero el pagefault que ahora no deberia generarlo. De lo contrario procederemos de la misma forma que para las demas excepciones del procesador, matamo el lemming y saltamos a la idle.

Entonces como funciona el handler del page fault. Lo primero que haremos es chequear que la dirección virtual a la que se quizo acceder y genero el falló se encuentre en el área designada para la memoria compartida entre los lemmings. De no ser así retornará falso y no realizara un mapeo. Si se encuentra en el rango, entonces vamos a mapear esta dirección. Pero antes debemos ver si otros lemmings del mismo equipo ya la mapearon antes. Para ello tenemos dos arreglos que guardan la direcciones fisicas mapeadas en el indice que se corresponde con la dirección virtual. Para saber que indice corresponde a que dirección virtual hacemos la siguiente codificación: $virtAddr = 0x400000 + i * 0x1000 \rightarrow i = (virtAddr - 0x400000)/0x1000$ La lógica de esto es que la base del area compartida es la dirección

0x400000 y todas las otras posibles direcciones mapeadas serán páginas de 4kib que estén una i cantidad de paginas adelante. Entonces una vez decodificado el indice del arreglo, obtenemos el índice del lemming que solicita el mapeo, y dependiendo su paridad será que arreglo miremos, si el del equipo Amalin o Betarote. A partir de la dirección virtual, chequeamos si tienen una dirección física válida(distinta de 0). Si lo tienen entonces mapeamos a esta misma, si no, pediremos una nueva pagina del área libre de usuario y mapearemos allí, agregando esta dirección física al arreglo del equipo que corresponde en el índice correspondiente. Por último devolvemos true.

2.7.6. n.

En game_tick() se llama a la función check_win_condition(), que recorre los 40 pixeles de ambos bordes verticales en busca de lemmings. En caso de que un leming del equipo 'B' esté en el borde izquierdo, gana B y al revés con el borde derecho. Se imprime en pantalla el equipo ganador y se llama a la función endGame(), que setea la variable debugging del scheduler en 2 que aprovecha el mecanismo utilizado en el mecanismo de debugging (que cuando .debugging!=0 detiene la ejecución, en concordancia con lo explicado para el punto 7.g.) para trabar el juego.