DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

SO

Organización del Computador II

Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Federico Ivan Barrena Guzman	236/18	fedebarrena@gmail.com
Camila Rosario Camaño	310/17	camicam26@gmail.com
Tomás Ossa	752/19	ossatomas2@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

El objetivo de este trabajo práctico es implementar paso a paso un pequeño juego para aplicar los conceptos de *System Programming* vistos en las clases teóricas y prácticas. Utilizaremos como entorno de pruebas el programa *Bochs* para simular una computadora IBM-PC.

Los archivos utilizados se encuentran ordenados de la siguiente manera:

- Dentro de la carpeta **src**:
  - `game.h, game.c` correspondientes con implementación de la lógica del juego,
  - `screen.h, screen.c` con rutinas para pintar la pantalla,
  - `map.h, map.c` con rutinas para el mapa,
  - `syscall.h` con una interfaz para los llamados a sistema,
  - `kernel.asm` con el código del kernel.
  - las carpetas `common`,`interrupt`,`memory` y `tasks`.
- Dentro de la carpeta **common** definiremos funciones y recursos utilizados a lo largo del juego:
  - `colors.h, defines.h` con constantes para los colores del mapa y Lemmings y definiciones varias,
  - `i386.h` con funciones auxiliares para escribir código Assembler desde C,
  - `kassert.h` con rutinas para garantizar invariantes en el kernel,
  - `types.h` con las estructuras comunes utilizados en el kernel,
  - `utility.h, utility.c` con funciones generales utilizadas a lo largo del juego.
- Dentro de la carpeta **interrupt**:
  - `idt.h, idt.c` con entradas para la IDT y funciones asociadas,
  - `isr.h, isr.asm` con definiciones de las rutinas de atención de interrupciones,
  - `handlers.c` con las implementaciones de las rutinas de interrupciones especiales,
  - `pic.h, pic.c` con las funciones `pic_enable, pic_disable, pic_finish1, pic_reset`.
- Dentro de la carpeta **memory**,
  - `a20.asm` con rutinas para habilitar y deshabilitar A20,
  - `gdt.h, gdt.c` con la definición de la tabla de descriptores globales,

    - `mmu.h, mmu.c` con las rutinas asociadas a la administración de memoria.

- Dentro de la carpeta **tasks**,

    - `idle.asm` con el código de la tarea *Idle*,
    - `sched.h, sched.c` con las rutinas asociadas al *scheduler*,
    - `tss.h, tss.c` con las definición de entradas de TSS,
    - `taskLemmingA.c, taskLemmingB.c` con los códigos de las tareas del jugador A y B respectivamente.

Los contenidos del informe son los siguientes:

- **Sección 2:** desarrollo y explicación de las estructuras utilizadas.
- **Sección 3:** explicación de los pasos de inicialización del sistema.
- **Sección 4:** explicación de las interrupciones implementadas y sus handlers.
- **Sección 5:** explicación del comportamiento del scheduler.
- **Sección 6:** explicación del funcionamiento del juego.

2. Estructuras

2.1. GDT

En los archivos `memory/gdt.h` y `memory/gdt.c` se encuentran las estructuras y variables que vamos a usar para acceder y administrar la **GDT** en nuestro sistema.

Según se indica en el enunciado del trabajo práctico, las primeras ocho posiciones de la tabla se encuentran reservadas. Por este motivo, definimos los segmentos que va a utilizar el sistema a partir de la posición nueve en la tabla (índice ocho).

Siguiendo las instrucciones del primer ejercicio, definimos **cuatro descriptores de segmento** que direccionan los primeros `817MiB` de memoria: uno para código de nivel 0, uno para datos de nivel 0, uno para código de nivel 3 y otro para datos de nivel 3 respectivamente. Para todos los casos, la base del segmento es el primer `byte` de memoria (`0x00000000`), el bit P (present) está prendido, el `bit` de granularidad G está prendido para que el límite se lea como múltiplo de `4KiB` y el límite es `0x330ff`: `817MiB` equivalen a `0x33100000 bytes`, entonces dividimos por `0x1000 (4KiB)` y restamos uno para obtener el límite. Indicamos que son segmentos de 32 bits prendiendo el bit de D/B y apagando el bit L y dejamos siempre el bit disponible AVL en 0. Indicaremos que son segmentos de código o datos prendiendo el bit S.

Para los dos **segmentos de código**, el valor asignado en el campo tipo es `0xa` (10) que indica que se trata de un segmento de código con permiso de lectura (y ejecución). Para los **segmentos de datos**, el tipo es `0x2`, indicando que son segmentos de datos con permisos de lectura y escritura. Por último, el DPL del segmento de código de nivel 0 y el de datos de nivel 0 es 0, mientras que el DPL para los dos segmentos restantes es 3.

La próxima entrada en la GDT corresponde al descriptor de segmento definido para completar el ejercicio 1.c: un segmento de datos con permiso de escritura (tipo = 0x2) de nivel 0 (DPL = 0) que tiene base en la dirección `0xa0000`, el inicio del espacio de memoria de video y límite `0x1ffff` con granularidad 0: de esta forma, se podrá acceder hasta el último `byte` de la memoria de video `0xbffff`.

A partir de la próxima entrada (índice 13), definimos los **descriptores de TSS** para las tareas inicial y Idle. La base de los segmentos es temporalmente `0x0`, se pisarán luego en tiempo de ejecución cuando el *kernel* indique que se deben inicializar los descriptores de estas tareas. El límite es el tamaño del `struct` que usamos para representar los TSS (que es equivalente a la estructura en memoria de un TSS). Los DPL son 0 porque únicamente podemos acceder con permisos de administrador y el tipo `0x9` junto con el bit S apagado indican que se trata de un descriptor de TSS. El resto de los bits se apagan excepto por el bit present según lo indica el manual.

En la entrada siguiente al descriptor de TSS de la tarea Idle definimos **10 descriptores de TSS** contiguos que van a ser utilizados para las tareas Lemming. El primero de ellos tendrá entonces índice 15. Se definen de manera similar a los descriptores de TSS anteriores, con la salvedad del bit present que apaga.

Como veremos más adelante, la cantidad máxima de tareas Lemming que pueden estar ejecutándose son 10, y cada una tendrá un índice o `id` entero del 0 al 9: el descriptor de TSS asociado a una de las tareas será el de la misma posición que el índice de la tarea contando desde el primer descriptor, es decir, el descriptor de TSS para una tarea Lemming tiene índice `15 + id`.

2.2. IDT

Vamos a definir las estructuras y funciones asociadas a la **IDT** en los archivos `interrupt/idt.h` e `interrupt/idt.c`.

Siguiendo las instrucciones del segundo ejercicio, iniciamos las 256 entradas de la IDT con todos sus atributos en 0, y completamos el descriptor de la IDT. Este último tiene su dirección base apuntando a la posición de memoria en donde se encuentra el `struch` que utilizaremos para la IDT, y límite como su tamaño menos 1.

En `idt_init` inicializamos las entradas 0 a 19 de la IDT correspondientes a las interrupciones y excepciones del procesador, y las entradas 32 y 33 para las rutinas de atención de reloj y teclado respectivamente. Son *interrupt gates* de nivel 0 (DPL = 0) porque queremos que sean atendidas por código con permisos de administrador. A su vez, completamos su offset con la dirección de comienzo de cada rutina, su selector de segmento apuntando al segmento de código de nivel 0, y el `bit` P en 1.

Definimos las entradas 88, 98 y 108 correspondientes a los servicios `move, explode` y `bridge`. Son *interrupt gates* de nivel 3 (DPL = 3) completadas con offset apuntando al lugar donde comienza cada rutina correspondiente, el selector de segmento de código de nivel 0, y el bit de presente en 1.

2.3. TSS

Definimos las estructuras necesarias para las TSS y sus funciones asociadas en los archivos `tasks/tss.h` y `tasks/tss.c`.

El TSS para la **tarea inicial** lo completamos con 0: ya que nunca se va a realizar el cambio de tarea por la tarea inicial, su TSS no necesita tener datos válidos.

El TSS de la **tarea Idle** lo completamos en tiempo de ejecución dentro de la función `tss_init_idle`. Idle es una tarea que ejecuta con permisos de administrador por lo tanto sus selectores de segmento de código y datos apuntan a los segmentos de código y datos de nivel 0. La tarea tiene la misma pila y el mismo directorio de páginas del kernel y los flags con las interrupciones habilitadas.

Para los TSS de las **tareas Lemming**, definimos un arreglo de 10 posiciones indexado por `id` de Lemming. La TSS de cada tarea se inicializa en la función `tss_init_lemming_task` cuando el sistema crea una nueva tarea Lemming.

En el registro `cs` se carga el selector de segmento de código de nivel 3 (RPL = 3), en el resto de selectores de segmento cargamos el selector de segmento de datos de nivel 3 (RPL = 3).

En el selector de segmento para la pila de nivel 0 cargamos el selector de segmento de datos de nivel 0, mientras que el `esp` de nivel 0 se calcula en tiempo de ejecución: se reserva una página libre del kernel para cada tarea Lemming creada. El `eip` está preestablecido en `0x8000000`, así como la base de la pila en la dirección virtual `0x8003000`. Al igual que en la tarea Idle, en los flags indicamos que las interrupciones están habilitadas.

En función del equipo al que pertenece la tarea Lemming siendo creada se decide la dirección física del comienzo del código de la tarea: si el Lemming es del equipo Amalin la dirección física es `0x18000`, si es de Betarote es `0x1a000`.

Una vez obtenida la dirección física de comienzo del código, cargamos en el `cr3` el puntero al directorio de tablas de página creado por la función `mmu_init_task_dir`. Esta función mapeará los primeros `4MiB` con *identity mapping* (área del kernel y área libre del kernel), las páginas del código a la dirección física calculada y la página para la pila a una página del espacio de memoria libre del usuario.

3. Inicialización del Kernel

En esta sección describiremos cómo es el proceso de iniciación del sistema implementando en `kernel.asm`.

3.1. Segmentación y activación modo protegido

Primero iniciamos el kernel en **modo real**, codificando las instrucciones en 16 bits. Se deshabilitan las interrupciones con `cli` y cambiamos el modo de video a 80 columnas y 50 filas. Imprimimos un mensaje de bienvenida y habilitamos A20 para poder direccionar más del primer `1MiB` de memoria. Luego, cargamos la GDT con `lgdt` y activamos el bit de *protection enable* del registro `cr0`. Finalmente, saltamos a modo protegido.

Una vez en **modo protegido**, empezamos a codificar las instrucciones en 32 bits. En este modo la GDT ya está completa y cargada. Definimos los selectores de segmentos `ds,es,fs,gs` como selectores de segmentos de datos de nivel 0 y `ss` como selector de segmento de pila. Establecemos en `ebp, esp` la base de la pila desde la dirección física 0x2500. Imprimimos un mensaje de bienvenida e iniciamos la pantalla con `screen_init_fs` desde Assembler. Luego, pasamos a la parte de activación de paginación.

3.2. Paginación

Inicializamos las estructuras necesarias para administrar la paginación llamando a la función `mmu_init`. Esta función, implementada en `memory/mmu.c`, únicamente se encarga de inicializar los punteros a las próximas páginas libres tanto del kernel como del espacio de memoria libre para ser usada por las tareas de nivel usuario.

La función `mmu_init_kernel_dir` realiza el mapeo de páginas para el kernel. Inicializa en el directorio de páginas a partir de la dirección prefijada `0x25000` y completa la primera tabla de página a partir de la dirección `0x26000`, mapeando así los primeros `4MiB` con *identity mapping*. Tanto para la PDE como todas las PTE de la primera tabla de página el bit de present está prendido, así como el permiso de escritura con el bit R/W. El bit de user/supervisor está en 0, indicando que sólo código privilegiado puede acceder.

La función retorna la dirección física en la que comienza el Page Directory, que en este caso coincide con el valor que cargamos en el `cr3`. Los 12 bits menos significativos del la dirección son nulos, en el `cr3` corresponden a bits de control que por defecto queremos que tengan valor nulo.

Una vez cargado entonces el `cr3` con el valor de retorno de esta función, habilitamos el bit de paginación del registro `cr0`. Luego, inicializamos la TSS.

3.3. Tareas

Una vez que activamos el sistema de paginación, preparamos las estructuras para administrar las tareas que van a ejecutarse.

La función `tss_init` en el archivo `tasks/tss.c` inicializa los TSS de la tarea inicial y todas las tareas Lemming con toda la estructura en 0. Como explicamos en la sección 2.3, para el TSS de la tarea inicial no necesitamos valores válidos porque nunca va a ejecutarse y los TSS de las tareas Lemming se completarán a medida que se creen las tareas.

La función `tss_init_idle` en el mismo archivo completa el TSS de la tarea Idle como especificado en la sección 2.3. Además, carga en la entrada de la GDT correspondiente la base del descriptor de TSS, es decir la dirección donde comienza la estructura.

Antes de pasar a ejecutar el código de la tarea Idle, inicializamos las estructuras del *scheduler* con la función `sched_init` de `tasks/sched.c` cuya lógica se detalla en la sección 5. Se inicializa también el juego con `game_init` (ver sección 6).

Preparamos el sistema para la atención de interrupciones preparando la IDT con `idt_init`, cargando en la tabla los datos especificados en la sección 2.2 y cargando el descriptor de la IDT con la instrucción `lidt`. Por último, se configura el PIC (1 y 2) con `pic_reset` y luego se habilita para que las instrucciones provenientes del PIC puedan ser atendidas una vez que se habiliten las interrupciones.

Finalmente, se carga en el `tr` el descriptor de TSS de la tarea inicial, para luego realizar el primer *task switch* por la tarea Idle. A partir de este momento, el sistema ya se encuentra inicializado y empieza a ejecutarse el código de la tarea Idle con interrupciones habilitadas (por los flags cargados en su TSS).

4. Interrupciones

En los archivos `interrupt/isr.h`, `interrupt/isr.asm` y `handlers.c` definimos e implementamos las rutinas de atención de interrupciones internas y externas.

Las **interrupciones 0 a 19** (menos la 14, excepción *page fault*) serán atendidas de la misma manera: dentro de su rutina en Assembler, llamamos a la función `handle_exception`. Esta función es la encargada de imprimir en pantalla el contexto de ejecución en el que fue interrumpido el sistema en caso de estar activado el modo debug. A su vez, llama a función `game_inform_exception`, la cual se encarga de desalojar la tarea actual que haya causado la excepción (siempre y cuando no sea un fallo de página) . La tarea es desalojada, por lo tanto no retorna a la rutina.

La rutina de excepción para **fallos de página** o *pagefaults* será diferente. Según la funcionalidad pedida, las tareas Lemming pueden acceder a memoria compartida que es mapeada por demanda. Es decir, se mapea únicamente cuando la tarea intenta acceder a ese espacio de memoria (si alguna vez lo hace). Además, si alguna tarea Lemming del mismo equipo ya tiene mapeada esa página, se debe replicar el mismo mapeo. En caso de ser una dirección fuera del rango del espacio para páginas de tareas, la tarea es desalojada. Para ello, utilizamos la función `handle_pagefault`. Esta función se encarga de resolver el problema de la siguiente manera:

1. En caso de estar en modo debug, imprime en pantalla la información necesaria. A su vez, le avisa al juego que hubo una excepción de tipo *page fault* con la función `game_inform_exception`.
2. En el registro `cr2` va a estar la dirección lineal a la que está intentando acceder la tarea actual. Verificamos que la dirección a la que la tarea intenta acceder es válida, o sea se encuentra entre la dirección virtual `0x400000` y `0x13FFFFF` como indica el enunciado. Si no lo es, informamos al juego que una tarea está intentando hacer un acceso inválido a memoria con la función `game_invalid_shared_memory_access`, que desalojará la tarea Lemming.
3. Si la dirección es válida, entonces mapeamos la página que corresponde con la ayuda de la función `tss_map_shared_page`. El primer paso es establecer si alguna tarea Lemming del mismo equipo que la tarea que generó la excepción tiene mapeada la dirección virtual que generó el problema. De esto se encarga la función `tss_team_mapped_addr`: por cada tarea dentro del equipo (excluyendo la actual), se hace un llamado a la función `mapped_addr`, que recibe el id de una tarea y una dirección virtual. Dentro de esta función, se verifica si la tarea está activa mirando el bit *present* dentro de la entrada de la GDT que le corresponde al TSS de la tarea, y finalmente verificando si la dirección virtual se encuentra mapeada, tomando el `cr3` de la tarea y accediendo a la PDE y PTE correspondientes a la dirección virtual. Si los bits de *present* se encuentran todos prendidos, entonces la página existe y devolvemos la dirección física en la que está mapeada la dirección. Si alguna de estas verificaciones falla, devolvemos la dirección `0x00000000`, que es considerada

inválida. Volviendo a `tss_map_shared_page` si obtuvimos una dirección física válida mapeamos la dirección virtual a esa dirección. En caso contrario, pedimos una página libre del área de memoria libre para las tareas de nivel usuario. En ambos casos, el mapeo se realiza con la ayuda de la función `mmu_map_page` y los permisos para la página son de escritura con nivel de privilegio de usuario (y *present)*.

Para la rutina de **excepción de reloj**, primero preservamos los registros con `pushad`, avisamos al pic que se recibió la interrupción con `pic_finish1`. Luego imprimimos el reloj del sistema en la pantalla del juego, y con `game_tick` avanzamos el reloj del juego. En la sección 6 explicaremos qué hace esta función en más detalle. Esta función devuelve el selector de TSS de la próxima tarea a ejecutarse o 0 si no debe haber un intercambio de tarea. Como es un error realizar un *task switch* con la misma tarea que se está ejecutando, comparamos el selector con el selector de la tarea actual, y en caso de ser igual no cambia. `popad, iret.`

En la rutina de **excepción de teclado** cargamos en el registro `eax` el scan code de la tecla. La función `handle_keyboard_int` contiene la lógica real de la rutina. Si la tecla presionada es *y*, se informa al juego que se intenta activar o desactivar el modo debug con la función `game_toggle_debug_mode`, explicada en la sección 6. Si la tecla es un número entre 0 y 9, se imprime el dígito en pantalla (esquina superior derecha), según requerido por el ejercicio 3c. Una vez ejecutada `handle_keyboard_int`, la rutina le avisa al pic que se recibió la interrupción, y termina con `popad, iret`.

El sistema cuenta con **tres servicios que las tareas Lemming** tienen disponibles: `move, explode, bridge`, y cada uno tiene un handler especial.

Tanto el servicio `explode` como `bridge` tienen la particularidad de que la tarea que usa el servicio es desalojada. Por este motivo, los servicios no preserva los registros y hacen llamados a funciones implementadas en C, que nunca retornan.

Para el caso del servicio `move`, se preservan todos los registros menos `eax` como especifica el enunciado, en donde se devuelve el resultado de la operación. Por este motivo, no utilizamos la instrucción `pushad`. Además, únicamente guardamos en la pila los registros `edi`, `esi`, `ecx`, `edx` porque las subrutinas que se llaman desde la interrupción respetan la convención C.

El *handler* delega la lógica a la función `move` implementada en C y luego realiza un cambio de tarea a la tarea Idle llamando a `switch_to_idle`, respetando la funcionalidad pedida.

El funcionamiento interno de las funciones implementadas en C es explicado en la sección 6.

5. Scheduler

Nuestro sistema va a utilizar un scheduler de tareas básico *round robin*, en donde por cada tick de reloj intercambia tareas de un equipo a otro o pasa a la tarea Idle en caso de haber ocurrido una excepción/syscall.

Las tareas Lemming tienen un identificador único numérico, que renombramos como el tipo `task_id_t`. Como el máximo de tareas Lemming que pueden estar activas en simultáneo es 10, el id es un número entre 0 y 9. Además, si el id es menor o igual que 5, el Lemming pertenece al equipo A. En caso contrario, el Lemming es del equipo B.

Basándonos en el gráfico del scheduler del enunciado, el comportamiento del scheduler será el siguiente:

- Si se está ejecutando una **tarea del equipo** A, luego del próximo ciclo de clock va a cambiar a una tarea del equipo B, y viceversa.

- Si ocurre una **syscall** `move`, luego de ejecutarla y después del próximo ciclo de clock pasa a la tarea Idle. Después de otro ciclo de clock, pasa a la próxima tarea Lemming. La tarea Lemming que realizó la llamada al servicio no es desalojada del scheduler.
- Si ocurre una **syscall** `explode`, primero se desalojan la tarea Lemming que la llamó y las tareas Lemmings que estén a su alrededor. Luego, pasa a la tarea Idle y en el próximo ciclo de clock busca la próxima tarea Lemming a ejecutar.
- Si ocurre una **syscall** `bridge`, la tarea que llamó al servicio es desalojada y el scheduler salta a la tarea Idle.
- Si ocurre una **excepción**, desaloja la tarea que la provocó (si es un pagefault la desaloja si accedió a una dirección lineal inválida, sino no). Pasa a la tarea Idle.

En los archivos `tasks/sched.h` y `tasks/sched.c` definimos todas las funciones necesarias para el scheduler.

El *scheduler* utiliza las siguientes variables para describir su estado:

- `running_task_id`: guarda el id de la tarea Lemming ejecutándose actualmente.
- `last_executed_A` y `last_executed_B`: índices de la última tarea Lemming del equipo A/B ejecutada.
- `lemming_A_tasks` y `lemming_B_tasks`: arreglos de valores booleanos por equipo, los cuales indicarán con *true* que el Lemmings con el id correspondiente a su índice dentro del arreglo está vivo (para el caso del equipo B, el índice más 5 coincide con el id del Lemming). Son de tamaño 5 cada uno, ya que es la máxima cantidad de Lemmings que puede tener un equipo.

Las funciones que implementa son:

- `get_running_task_id`: devuelve el id de la tarea Lemming que se está ejecutando.
- `sched_init`: inicializa las variables `running_task_id` con 9, `last_executed_A` y `last_executed_B` con 4, para empezar a ejecutar desde el primer Lemming del equipo A cuando se carguen tareas Lemming en el *scheduler*. Luego, iniciamos todos los Lemmings como inactivos en `lemming_A_tasks` y `lemming_B_tasks`.
- `sched_init_task`: toma un equipo de Lemmings como parámetro y retorna el id de la tarea creada. Antes de comenzar, consigue el índice de la próxima tarea libre del equipo con `sched_next_free_task`. Si el equipo ya tiene la máxima cantidad de Lemmings activos, devuelve un índice inválido. Si hay lugar para la nueva tarea, se inicializa el TSS según describimos en la sección 2.3 con la ayuda de la función `tss_init_lemming_task`. Luego, se actualiza el arreglo de booleanos que le corresponde al equipo para indicar que ahora una nueva tarea está activa y se actualiza la entrada de la GDT (bit de *present* y dirección base del TSS) que le corresponde al descriptor de TSS de la tarea , que como explicamos en la sección 2.1 tiene índice `15 + id`.
- `sched_next_task`: devuelve el selector de TSS de la próxima tarea que tiene que ser ejecutada. Para ello, consigue el índice del próximo lemming que debe ser ejecutado. Chequea si es un índice válido, actualiza `running_task_id` con el índice encontrado y `last_executed_A` o `last_executed_B` según corresponda. Finalmente, calcula el índice en la GDT que le corresponde al descriptor de TSS de la nueva tarea a ser ejecutada. A partir del índice, el selector se calcula con un *shift* a derecha de 3 *bits*, indicando que el descriptor se encuentra en la GDT y que el RPL es 0.
- `sched_kill_task`: se encarga de desalojar una tarea. En nuestro sistema, va a desactivar el bit de presente en su descriptor de TSS en la GDT. Luego, va a indicar en el arreglo de Lemmings correspondiente a su equipo que se encuentra muerto.
- `sched_kill_running_task`: va a desalojar la tarea actual con `sched_kill_task`, y pasar a la tarea Idle.

- `sched_kill_all`: se encarga de desalojar todas las tareas Lemmings. La tarea corriendo actualmente es la última en ser desalojada.
- `all_dead`: retorna un booleano que indica si todos los Lemmings del juego están muertos.
- `get_next_in_team`: toma un equipo de Lemmings como parámetro y retorna el índice del próximo Lemming que será ejecutado en ese equipo. Para ello, sumamos 1 a `last_executed_A` o `last_executed_B` según corresponda. Luego, vamos a buscar su estado en su arreglo de equipo de Lemmings. Devolvemos el primer id de tarea que se encuentre activa. En caso de no encontrar ninguna activa, el valor devuelto corresponderá con el `last_executed_{team}`.
- `get_next_lemming_id`: se encarga de devolver el índice del próximo Lemming a ejecutar. Primero, verifica si todos los Lemmings están inactivos, de ser ese el caso, devuelve 10. Si no, comienza a buscar la próxima tarea Lemming a ejecutar. Si el equipo de la tarea actual es A, entonces busca el próximo a ejecutar en el equipo B y a su vez chequea que esté vivo. Si no está vivo, consigue el próximo lemming del equipo A. En caso de que la tarea actual sea B, realiza lo mismo pero con el próximo a ejecutar del equipo A. Finalmente, retorna el índice de la próxima tarea encontrado.
- `sched_next_free_task`: dado un equipo retorna el próximo índice libre para crear una tarea Lemming de ese equipo. Devuelve el primer id en orden que pertenece a una tarea inactiva utilizando el arreglo de booleanos correspondiente. En caso de que no haya lugar, retorna 10 para indicar que no hay espacio.

6. Lógica del juego

La **lógica del juego** se implementa mayormente en el archivo `game.c`, exceptuando la lógica de actualización y manejo del mapa.

6.1. Mapa

El **mapa del juego** ocupa las primeras 80 filas y 40 columnas de la pantalla. Debajo del mapa, se encuentran los datos de cada equipo con los Lemmings activos y la cantidad de Lemmings creados en total para cada equipo. Los archivos `map.h` y `map.c` tienen el código para actualizar e inicializar el mapa.

El mapa se representa con una matriz de caracteres de 80 filas y 40 columnas. Dentro de esta matriz, los valores posibles son el carácter '.', que representa un espacio libre en el mapa, 'P' que representa una pared, 'L' para representar agua, 'X' que marca una pared rota y '+' para puentes. A medida que las tareas Lemming vayan interactuando con el mapa se actualizará la matriz.

Las funciones principales que se exponen son:

- `map_reset`: inicializa el mapa del juego. Copia el mapa original, del comienzo del juego a la matriz que mencionamos.
- `map_print`: imprime en pantalla todo el mapa. Por cada posición dentro del área del mapa, se imprime en pantalla una representación de los distintos elementos posibles.
- `map_explosion`: actualiza el mapa teniendo en cuenta que hubo una explosión en la posición que se recibe por parámetro. Por cada posición adyacente a la posición, si había una pared se actualiza la matriz del mapa con una pared rota y se refleja el cambio en pantalla, imprimiendo en la posición de la pared la representación de la pared rota.
- `map_bridge`: recibe una posición donde se quiere crear un puente y actualiza el mapa si en esa posición había agua. Como con `map_explosion`, se actualiza la matriz y la pantalla únicamente en esa posición.

- `map_spawn_lemming`: dada una posición y un equipo Lemming, imprime en la posición un carácter que representa un Lemming. Las posiciones de los Lemming no se guardan en la matriz. El mapa no guarda referencias a las posiciones de los Lemmings.
- `map_move_lemming`: recibe un equipo, la posición anterior de un Lemming y su nueva posición. Imprime en pantalla el elemento que corresponda en la posición anterior y en la nueva posición el carácter que representa el Lemming.
- `map_kill_lemming`: recibe una posición donde murió un Lemming. Actualiza la pantalla con el la representación del elemento en esa posición en el mapa.
- `in_range`: devuelve *true* si la posición que recibe se encuentra dentro del rango del mapa.
- `is_free_space, is_wall, is_water, is_bridge, is_broken_wall`: que devuelven *true* si en la posición que reciben se tiene el elemento que indica el nombre de cada función respectivamente.

6.2. Inicialización del juego

El juego cuenta con varias variables para manejar el estado del juego:

- `debug`: booleano que indica si el juego se encuentra en modo debug (por defecto *false*).
- `exception_interrupted`: booleano que indica si alguna tarea Lemming generó una excepción. Se reinicia al valor por defecto *false* cada vez que el usuario decide activar o desactivar el modo debug.
- `game_finished`: booleano que indica si el juego terminó.
- `lemmings`: arreglo de tamaño 10 con la posición de cada Lemming, indexado por id de tarea. Consideramos que una tarea Lemming está activa si la posición dentro de este arreglo se encuentra dentro del rango del mapa.
- `ticks`: contador de cantidad de ciclos de reloj que pasaron desde el inicio del juego.
- `count_spawned_{team}, count_killed_{team}, alive_lemming_{team}`: contadores por equipo que indica la cantidad de Lemmings creados, la cantidad de Lemmings que fueron desalojados y la cantidad de tareas activas por equipo respectivamente.
- `order_spawned_{team}`: arreglos de 5 posiciones (una por Lemming dentro del equipo) de enteros que indican el orden en el que fueron creados los Lemming, indexado por id de tarea (para el equipo B, índice + 5 = id). El valor 0 en el arreglo representa una tarea inactiva, los valores numéricos menores representan una tarea más nueva y los valores más altos tareas con más tiempo en ejecución. En todo momento, se cumple que para las tareas activas el valor correspondiente en estos arreglos es único.

La **inicialización del juego** se realiza en la función `game_init`, que se encarga de imprimir en pantalla el mapa y los marcadores de los equipos con la ayuda de las funciones `map_print` y `screen_init_layout`. Además, inicializa los arreglos con las posiciones de los Lemmings en posiciones fuera del mapa y los arreglos con el orden de creación de los Lemmings con el valor 0.

6.3. Ciclos de reloj

La función `game_tick` se ejecuta cada vez que recibimos una interrupción de reloj, como vimos en la sección 4. Esta función devuelve el selector de TSS de la próxima tarea que debe ejecutarse. Por este motivo, lo primero que hacemos es verificar si el juego terminó o si hubo una excepción y el modo debug está activado (en este caso sabemos que el juego está interrumpido y en pantalla se está mostrando la información de debug. Si estamos en alguno de estos dos casos, entonces la tarea que debe ejecutarse es la tarea Idle. La función devuelve su selector de TSS.

Si no estamos en ninguno de estos casos, verificamos si pasaron 401 ciclos de reloj desde la creación del último Lemming. Si es así y la cantidad de Lemmings activos por equipo es menor que el máximo, se crea un Lemming con la función `spawn_lemming`. Cada 2001 ciclos de reloj, si un equipo tiene la cantidad máxima de Lemmings vivos, se desaloja el Lemming más antiguo y se crea uno nuevo con la función `respawn_lemming`. Finalmente, actualizamos el contador de ciclos de reloj, se actualiza en pantalla el reloj de la tarea Lemming que estaba ejecutándose y devolvemos el selector de TSS de la próxima tarea Lemming que debe ejecutarse con la función `sched_next_task`.

La función `spawn_lemming` crea un nuevo Lemming para un equipo dado, si la posición en la que debe crearse el Lemming no está ocupada (función `spawn_position_occupied`). Si es posible realizar la creación, se delega la creación de la tarea a `sched_init_task`. Luego, se actualizan todas las variables del estado del juego y se refleja en pantalla la creación del Lemming.

La función `respawn_lemming` tiene una función similar a `spawn_lemming`, con la diferencia que la nueva tarea tiene que reemplazar a la misma que estaba corriendo antes. Por esto, no actualizamos el *scheduler*, porque la tarea va a seguir activa, pero sí reiniciamos el TSS de la tarea con `tss_init_lemming_task`, porque queremos que empiece a ejecutar como si hubiera sido activada desde el inicio. Luego de esto, actualizamos las variables de estado del juego y la pantalla para ver reflejado el cambio.

6.4. Servicios del juego

El juego cuenta con tres servicios para las tareas lemming: `move, bridge` y `explode`. El comportamiento de cada una está definida en `game.c` y hacen lo siguiente:

- La syscall **move** recibe como parámetro en el registro `eax` la dirección a la que desea avanzar el Lemming. Retorna en `eax` el resultado del movimiento, en donde con 0 indicamos que pudo desplazarse sin problemas, y con los valores 1 a 4 qué encontró en la posición a la que quiso avanzar. La tarea no es desalojada. La función `move` describe lo siguiente:
  1. Primero verificamos si la dirección pasada por parámetro es un valor válido. De no ser así, desalojamos la tarea.
  2. Conseguimos el índice del Lemming actual que llama al servicio y la posición a la que desea avanzar.
  3. Chequea qué hay en la posición deseada. En caso de haber un borde, una pared, agua o un Lemming retornamos el valor correspondiente.
  4. Si no hay nada en la posición a la que quiere avanzar, actualizamos la posición del Lemming y el mapa.
  5. Luego, chequeamos si la nueva posición a la que avanzó cumple con los requisitos de finalización del juego (el Lemming llegó al extremo opuesto del mapa). De cumplirlos, el juego termina.
  6. Si el juego no termina, indicamos que el movimiento fue un éxito retornando 0.
- La syscall **explode** no recibe ni retorna ningún parámetro. El Lemming que invoca a esta llamada se autodestruye, destruir todas las paredes de ladrillo y a todos los Lemmings a su alrededor. La tarea es desalojada, al igual que las tareas afectadas. Explicaremos la función `explode`:
  1. Conseguimos el índice del Lemming actual que llama al servicio y su posición actual.
  2. Buscamos los Lemmings contiguos dentro del arreglo de Lemmings con `close`, y en caso de estar cerca, desalojamos la tarea Lemming con `kill_lemming`.
  3. Actualizamos el mapa para derribar paredes con `map_explosion`
  4. Desalojamos la tarea Lemming actual con `kill_lemming`.

- La syscall **bridge** recibe en `eax` la dirección a la que el Lemming desea crear un puente. Si en la posición indicada no hay agua, no se construye ningún puente. No retorna ningún valor y la tarea es desalojada. La función `bridge` realiza lo siguiente:
  - Conseguimos el índice del Lemming actual que llama al servicio y su posición.
  - Si la dirección pasada por parámetro es inválida, desalojamos la tarea.
  - Obtenemos la posición indicada por la dirección.
  - Con `map_bridge` creamos el puente en caso de que haya agua.
  - Finalmente desalojamos la tarea Lemming actual.

La función `kill_lemming` se encarga de actualizar el estado del juego. Actualiza la cantidad de Lemmings vivos, cantidad de Lemmings desalojados, la posición del Lemming (a una posición inválida) y el orden en el que fue creada la tarea a 0, indicando que ya no está activa. Actualiza la pantalla con `map_kill_lemming` y `screen_print_lemming_status`. Finalmente, le indica al *scheduler* que la tarea debe ser desalojada con la función `sched_kill_task` (o `sched_kill_running_task` si la tarea a ser desalojada es la tarea actual).

6.5. Modo Debug

El modo debug es un sistema especial que posee el juego con el propósito de mostrar en pantalla la primera excepción capturada y el estado del procesador en ese instante. Es activado cuando se presiona la tecla *y*. Luego de imprimir en pantalla la información necesaria, el juego se detiene hasta que la tecla *y* sea presionada nuevamente, en donde el modo debug se mantiene pero la información en la pantalla es eliminada. Al reanudar el juego, se espera hasta el próximo ciclo de reloj para decidir la próxima tarea a ser ejecutada. Vamos a indicar que el modo debug está activo con un cartel en la esquina inferior izquierda.

Este comportamiento lo implementamos de la siguiente manera:

- Si en una interrupción de teclado el scan code es el de la letra 'y', pasamos a `game_toggle_debug_mode`, una función en donde vamos a activar o desactivar modo debug. Se encuentra definida en `game.h` y `game.c`,
  - Si modo debug ya estaba activado y a su vez ocurre una interrupción, regresamos a la pantalla original y desactivamos modo debug.
  - Si modo debug no estaba activado, lo iniciamos.
  - Reiniciamos el indicador de excepción: exception_interrupted = false
  - Con la función `print_debug_mode`, imprimimos la información en pantalla si modo debug está activado.
- Por cada tick de reloj, en `game_tick` chequeamos si hubo una excepción y si modo debug está activo, pasa a la tarea Idle.
- En los handlers de interrupciones, vamos a chequear si modo debug está activo. En caso de estarlo, se imprime en pantalla toda la información necesaria con `print_exception`.

Los handlers de excepción implementados en C reciben un parámetro `exception_stack_t`, estructura definida en `common/types.h`. La estructura está preparada para recibir los valores de los registros de propósito general teniendo en cuenta que se ejecuta la instrucción `pushad` y algunos valores del stack de nivel 0. Estos valores son traducidos con la ayuda de la función `from_exception_stack`, que traduce la estructura `exception_stack_t` a `exception_debug_info_t`, siguiendo las reglas especificadas en el manual de los valores que tiene la pila dentro de una excepción cuando hubo un cambio de nivel de privilegio.

6.6. Finalización del juego

Las condiciones de finalización del juego solo pueden suceder en un `move` de alguna tarea lemming ya que para ganar, un lemming debe llegar al borde del equipo contrario. Entonces, vamos a verificar con la función `game_won` en cada `move` que se realiza si la posición a la que se está avanzando se encuentra en el borde del equipo contrario. Cuando el juego termina, imprimimos en pantalla el equipo ganador, la cantidad de lemmings creados por equipo en la totalidad del juego y desalojamos todas las tareas. A su vez, mantenemos un valor en memoria `game_finished` para conocer el estado del juego. El scheduler quedará corriendo la tarea Idle.