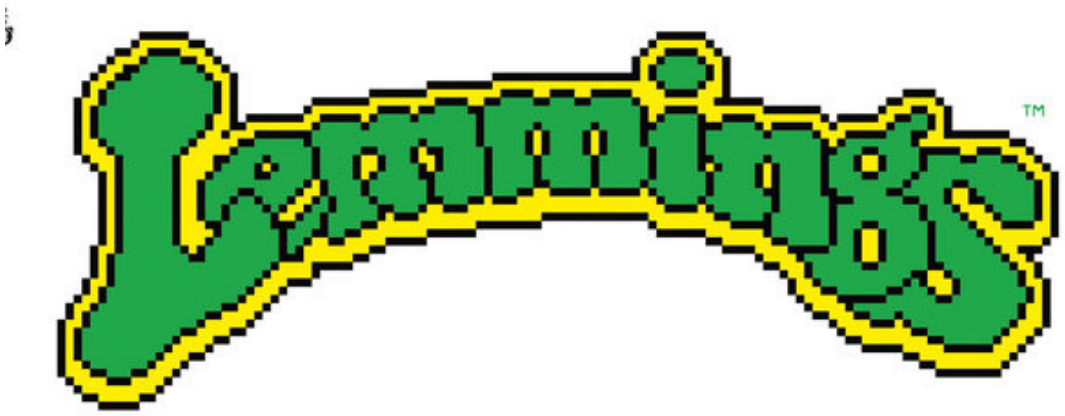
Trabajo Práctico 3

System Programming

Primer Cuatrimestre 2021

Forest Adventure



0.1 Objetivo

Este trabajo práctico consiste en un conjunto de ejercicios en los que se aplican de forma gradual los conceptos de *System Programming* vistos en las clases teóricas y prácticas. El objetivo es implementar un juego inspirado en el juego *«Lemmings»*[1].

Los Lemmings son curiosas criaturas a las que les encanta hacer *«trekking»*, sorteando obstáculos y recorriendo los terrenos más tenuosos. Además, tienen un gran espíritu competitivo.

Entre sus tradiciones más preciadas, se encuentra el juego de Tutunito, en el cual dos pueblos de Lemmings compiten en una carrera de obstáculos, con el objetivo de llegar a la «base» del pueblo opuesto. El primer pueblo cuyo Lemming llegue a la base opuesta, gana la competencia.

Los Lemmings se reproducen rápidamente, y suelen tener problemas de sobrepoblación. Teniendo en cuenta esto, los Lemmings tienen un gran sentimiento de comunidad y compañerismo: tienen una muy buena coordinación de equipo y consideran que es más importante ganar la competencia que su propia salud.

Si bien a los Lemmings les encanta recorrer y explorar, durante la competencia están vendados, lo cual les limita ver mucho más allá de lo que tienen directamente enfrente.

En la competencia hay dos tipos de obstáculos posibles: paredes y lagos. Cuando un Lemming se encuentra con un lago, este puede elegir usar su cuerpo como puente para que otros Lemmings puedan cruzar. Cuando un Lemming se encuentra con una pared, este puede elegir «explotar» para tirarla abajo.

Es importante recordar, que si bien este trabajo práctico esta ilustrado por un juego, los ejercicios proponen utilizar los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo.

[1] `https://es.wikipedia.org/wiki/Lemmings`

0.2 Introducción

Para este trabajo se utilizará como entorno de pruebas el programa *Bochs*. El mismo permite simular una computadora IBM-PC compatible desde el inicio, y realizar tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en *Bochs* de forma sencilla.

Una computadora al iniciar comienza con la ejecución del POST y el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En este caso dispondremos de un *Floppy Disk* como dispositivo de booteo. En el primer sector de dicho *floppy*, se almacena el *boot-sector*. El BIOS se encarga de copiar a memoria 512 bytes del sector, a partir de la dirección `0x7C00`. Luego, se comienza a ejecutar el código a partir esta dirección. El boot-sector debe encontrar en el *floppy* el archivo `KERNEL.BIN` y copiarlo a memoria. Éste se copia a partir de la dirección `0x1200`, y luego se ejecuta a partir de esa misma dirección. En la figura 2 se presenta el mapa de organización de la memoria utilizada por el *kernel*.

Es importante tener en cuenta que el código del *boot-sector* se encarga exclusivamente de copiar el *kernel* y dar el control al mismo, es decir, no cambia el modo del procesador. El código del *boot-sector*, como así todo el esquema de trabajo para armar el *kernel* y correr tareas, es provisto por la cátedra.

Los archivos a utilizar como punto de partida para este trabajo práctico son los siguientes:

- `Makefile` - encargado de compilar y generar el *floppy disk*.
- `bochsrc` y `bochsdbg` - configuración para inicializar Bochs.
- `diskette.img` - la imagen del *floppy* que contiene el *boot-sector* preparado para cargar el *kernel*.
- `kernel.asm` - esquema básico del código para el *kernel*.
- `defines.h` y `colors.h` - constantes y definiciones.
- `gdt.h` y `gdt.c` - definición de la tabla de descriptores globales.
- `tss.h` y `tss.c` - definición de entradas de TSS.
- `idt.h` y `idt.c` - entradas para la IDT y funciones asociadas como `idt_init` para completar entradas en la IDT.
- `isr.h` y `isr.asm` - definiciones de las rutinas para atender interrupciones (*Interrupt Service Routines*).
- `sched.h` y `sched.c` - rutinas asociadas al *scheduler*.
- `mmu.h` y `mmu.c` - rutinas asociadas a la administración de memoria.
- `screen.h` y `screen.c` - rutinas para pintar la pantalla.
- `a20.asm` - rutinas para habilitar y deshabilitar A20.
- `print.mac` - macros útiles para imprimir por pantalla y transformar valores.
- `idle.asm` - código de la tarea `Idle`.
- `game.h` y `game.c` - implementación de los llamados al sistema y lógica del juego.
- `syscalls.h` - interfaz a utilizar desde las tareas para los llamados al sistema.
- `kassert.h` - rutinas para garantizar invariantes en el kernel.
- `mapa.c` y `mapa.h` - mapa del juego.
- `taskLemmingA.c` - código de las tareas del jugador A.
- `taskLemmingB.c` - código de las tareas del jugador B.
- `i386.h` - funciones auxiliares para utilizar *assembly* desde C.
- `pic.c` y `pic.h` - funciones `pic_enable`, `pic_disable`, `pic_finish1` y `pic_reset`.

Todos los archivos provistos por la cátedra **pueden** y **deben** ser modificados. Los mismos sirven como guía del trabajo y están armados de esa forma, es decir, que antes de utilizar cualquier parte del código **deben** entenderla y modificarla para que cumpla con las especificaciones de su propia implementación. Tengan en cuenta que el código provisto por la cátedra puede **no** funcionar en todos los casos, y depende del desarrollo de cada uno de los trabajos prácticos.

A continuación se da paso al enunciado, se recomienda leerlo en su totalidad antes de comenzar con los ejercicios. El núcleo de los ejercicios será explicado en las clases, dejando cuestiones cosméticas y de informe para que realicen por su cuenta.

0.3 Historia

Los pueblos de Lemmings, pacíficos y trabajadores, rara vez entran en conflicto entre sí o con otros pueblos lindantes. Su economía se basa en labrar la tierra e intercambiar productos construidos con poca o nula maquinaria industrial. En su extensa cultura, existen ciertas tradiciones que comparten todos los pueblos Lemmings. Una de estas costumbres es el juego de *«Tutunito»*, que se remonta a los antiguos Lemmings originarios, por el año 1991 después del tercer sol. El juego de Tutunito clásico consistía en una carrera de velocidad por las tierras del valle de *Titinita* a orillas del rio *Umalote*. El recorrido comenzaba en el pueblo Lemming de *Amalin* y terminaba en el pueblo de *Betarote*. Si bien la tradición de los juegos se mantiene, las reglas del juego variaron significativamente desde las antiguas épocas. El juego dejó de ser una simple carrera de velocidad, para ser una carrera de supervivencia, con múltiples obstáculos. El objetivo se mantiene, pero ahora se juega en un área donde los jugadores corren con los ojos vendados topándose con paredes que deben demoler o lagos que deben cruzar. La propuesta de este tp consiste en reproducir este antiguo juego, pero en un tablero y reemplazando a los pobres lemmings de ojos vendados por tareas. Los lemmings deberán llegar desde un pueblo al otro (llegando al extremo opuesto del tablero), sin tener visibilidad más allá de lo que puedan tocar (a una celda de distancia).

0.4 Juego de Tutunito

El juego en sí consiste en dos pueblos de Lemmings, *Amalin* y *Betarote* que irán enviando sus Lemmings a recorrer el valle de *Titinita*, tratando de llegar al pueblo opuesto. Estos Lemmings saldrán en orden desde un punto predeterminado, con un máximo de 5 Lemmings en simultáneo por equipo. Si

ya hay 5 Lemmings en juego, luego de cierto tiempo, el Lemming más antiguo se da por vencido y le deja su lugar a uno nuevo.

El valle de *Titinita* será modelado como un mapa rectangular de 80x40 celdas, y va a contar con ciertos obstáculos: paredes, lagos, y los bordes del valle.

La lógica de los Lemmings será programada de antemano, y cada uno de estos empezará en la misma posición y podrá realizar una de tres acciones: moverse en una dirección, explotar, o hacer un puente.

La figura 1 muestra un ejemplo de como se presentará el juego en pantalla. Lamentablemente no tenemos control de una placa de vídeo y estamos programando en ASM y C, por lo tanto los gráficos resultantes serán un poco más rústicos que los presentados en la figura. No obstante, se presenta toda la información necesaria para el juego: los Lemmings ejecutandose, sus posiciones y el mapa.

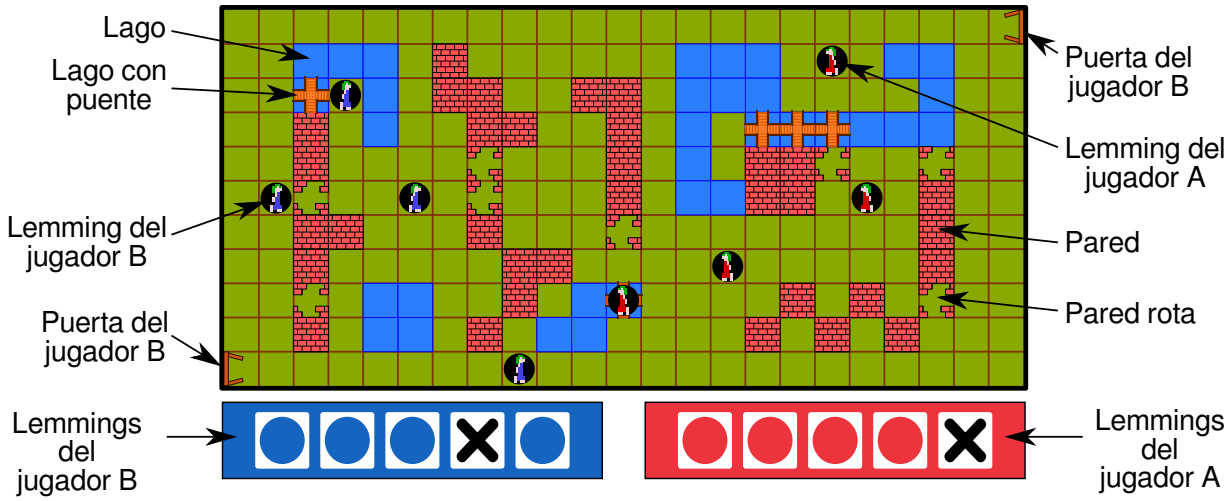


Figura 1: El Juego

El sistema ejecutará el código de los Lemmings como tareas independientes. Todas estas tareas comparten una página de código de 8KiB, en la que solo pueden leer. Tienen una página de lectura/escritura que usarán como pila y podrán comunicarse mediante un sistema de páginas compartidas que se asigna bajo demanda.

Cada 401 ciclos de reloj, si algún equipo tiene menos de 5 Lemmings en juego, el sistema creará un nuevo Lemming para ese equipo, posicionándolo en la dirección inicial del mapa para su equipo.

Cada 2001 ciclos de reloj, si algún equipo tiene 5 Lemmings en juego, el sistema desalojará al Lemming más antiguo y creará uno nuevo, posicionandolo en la dirección inicial del mapa para su equipo.

Si por algún motivo la dirección inicial del mapa para un equipo está ocupada, y esto no le permite crear el Lemming, la creación falla y no será creado un Lemming.

Una vez que son creados, los Lemmings pueden llamar a servicios del sistema para realizar algunas de las acciones que tienen disponibles, con el objetivo de llegar al extremo opuesto de la pantalla.

El juego termina cuando algún Lemming llega al extremo opuesto del mapa (el jugador de la izquierda gana cuando alguno de sus lemmings llega a cualquier celda en la última columna del mapa).

0.4.1 Tareas

El sistema dispondrá de tres servicios `move`, `explode`, y `bridge`. Estos se atenderán con un número de interrupción diferente para cada uno. Los parámetros de cada servicio se describen a continuación:

- Syscall `move` int 88

Parámetros	Descripción
`in EAX=direccion`	0: Arriba, 1: Derecha, 2: Abajo, 3: Izquierda
`out EAX=result`	0: Se desplazó sin problemas. 1: Pared de Ladrillo. 2: Agua. 3: Borde del mapa. 4: Lemming.

  Toma como parámetro de entrada en `EAX` una dirección de movimiento. El sistema revisará si puede mover el Lemming en esa dirección o hay algún obstáculo que lo impida. En caso de que no haya ningún obstáculo, el servicio retornará 0 en `EAX`.

  En caso de que haya un obstáculo, el sistema le retornará un código al Lemming en `EAX` para que este pueda decidir qué hacer al respecto. Un Lemming no podrá moverse pasando el borde del mapa, ni sobre agua, ni sobre una pared de ladrillos.

- Syscall `explode` int 98

Parámetros	Descripción
×	Esta llamada a sistema no toma parámetros.

  El Lemming que invoca a esta llamada a sistema se autodestruye, destruyendo todas las paredes de ladrillo y a todos los Lemmings que se encuentren a su alrededor. Esta syscall no retorna. El Lemming que la invoca es desalojado del sistema.

- Syscall `bridge` int 108

Parámetros	Descripción
`in EAX=direccion`	0: Arriba, 1: Derecha, 2: Abajo, 3: Izquierda

  Esta llamada a sistema puede ser usada por un Lemming para crear un puente. En `EAX` se indica la dirección con respecto a la posición del lemming en donde se desea crear el puente. En caso de que haya agua en esa posición, un puente será creado.

  Si no hay agua en esa posición, el puente no será creado.

  En cualquier caso, el Lemming que realizó la acción será desalojado del sistema.

Por otro lado, ninguno de los servicios debe modificar ningún registro, a excepción de los indicados anteriormente. Tener en consideración que luego del llamado a cualquiera de los servicios, la tarea en ejecución "pierde el turno". Es decir, que es desalojada del scheduler, y debe esperar hasta que pueda ser ejecutada nuevamente. En el tiempo entre que la tarea es desalojada y llega una interrupción de reloj para ejecutar la próxima tarea, el sistema debe ejecutar a la tarea `Idle`. Esta última, no tiene más propósito que mantener al procesador realizando alguna tarea, a la espera del próximo punto de sincronismo. Por último, si alguna de las llamadas a sistema recibe un argumento inválido, el sistema operativo debe eliminar la tarea, como si esta generara una excepción.

0.4.2 Organización de la memoria

El primer MiB de memoria física será organizado según indica la figura 2. En la misma se observa que a partir de la dirección `0x1200` se encuentra ubicado el *kernel*; inmediatamente después se ubica el código de las tareas LemmingA y LemmingB, y a continuación el código de la tarea Idle. El resto del mapa muestra el rango para la pila del kernel, desde `0x24000` a `0x25000` y a continuación la tabla y directorio de páginas donde inicializar paginación para el kernel. La parte derecha de la figura muestra la memoria a partir de la dirección `0xA0000`, donde se encuentra mapeada la memoria de vídeo y el código del `BIOS`.

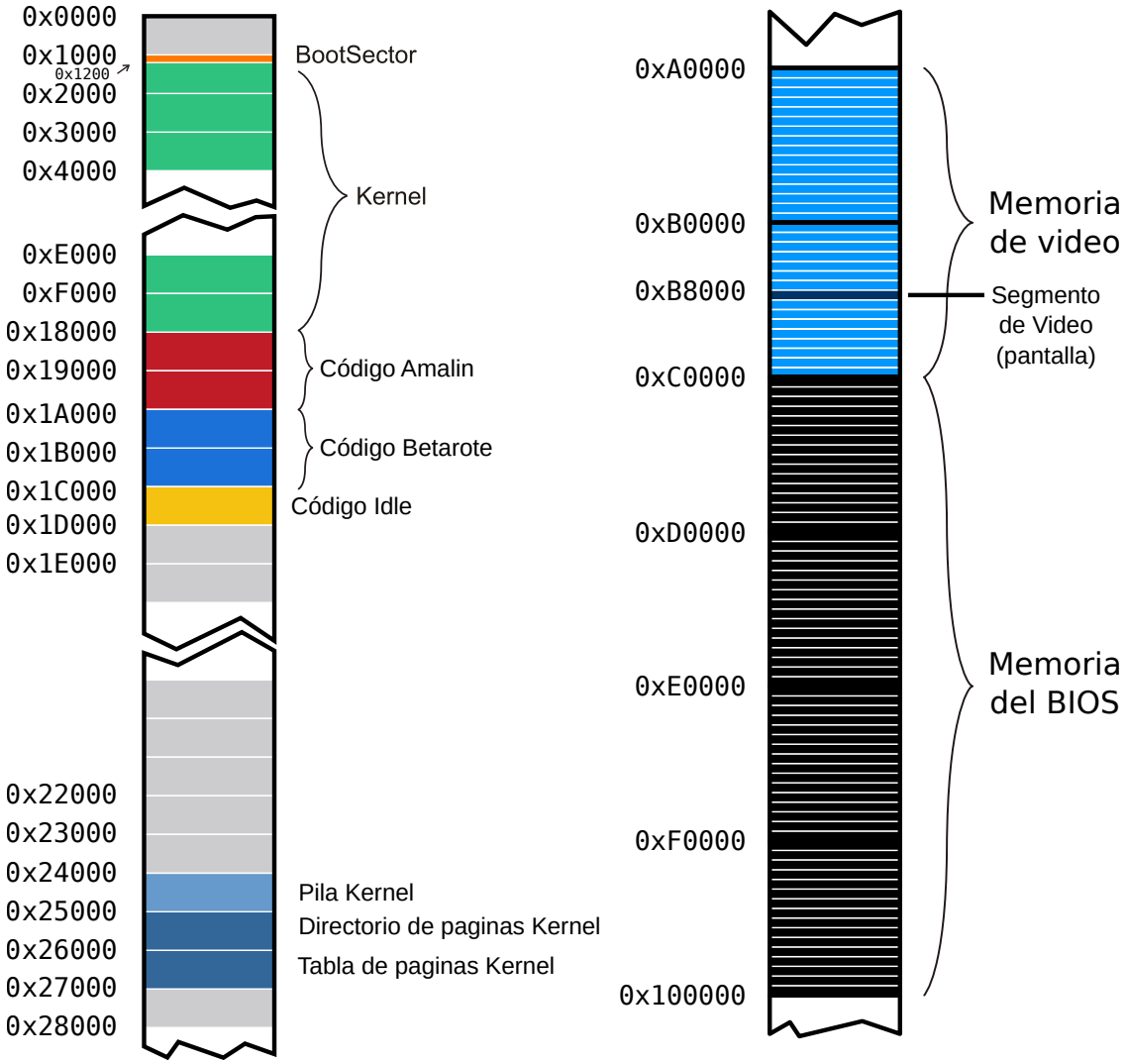


Figura 2: Mapa de la organización de la memoria física del *kernel*

En un rango más amplio, fuera del primer MiB, memoria física estará dividida en cuatro áreas: *kernel*, *área libre kernel* y *área libre tareas*

El área *kernel* corresponderá al primer MiB de memoria, el *área libre kernel* a los siguientes 3 MiB de memoria, y el *área libre tareas* a los siguientes 3 MiB de memoria.

La administración del área libre de memoria será muy básica. Se tendrá un contador de páginas a partir del cual se solicitará una nueva página. Este contador se aumentará siempre que se requiera usar una nueva página de memoria, y nunca se liberarán las páginas pedidas.

Las páginas del *área libre kernel* serán utilizadas para datos del kernel: directorios de páginas, tablas de páginas y pilas de nivel cero. Las páginas del *área libre tareas* serán utilizadas para datos de las tareas: stack y memoria compartida bajo demanda.

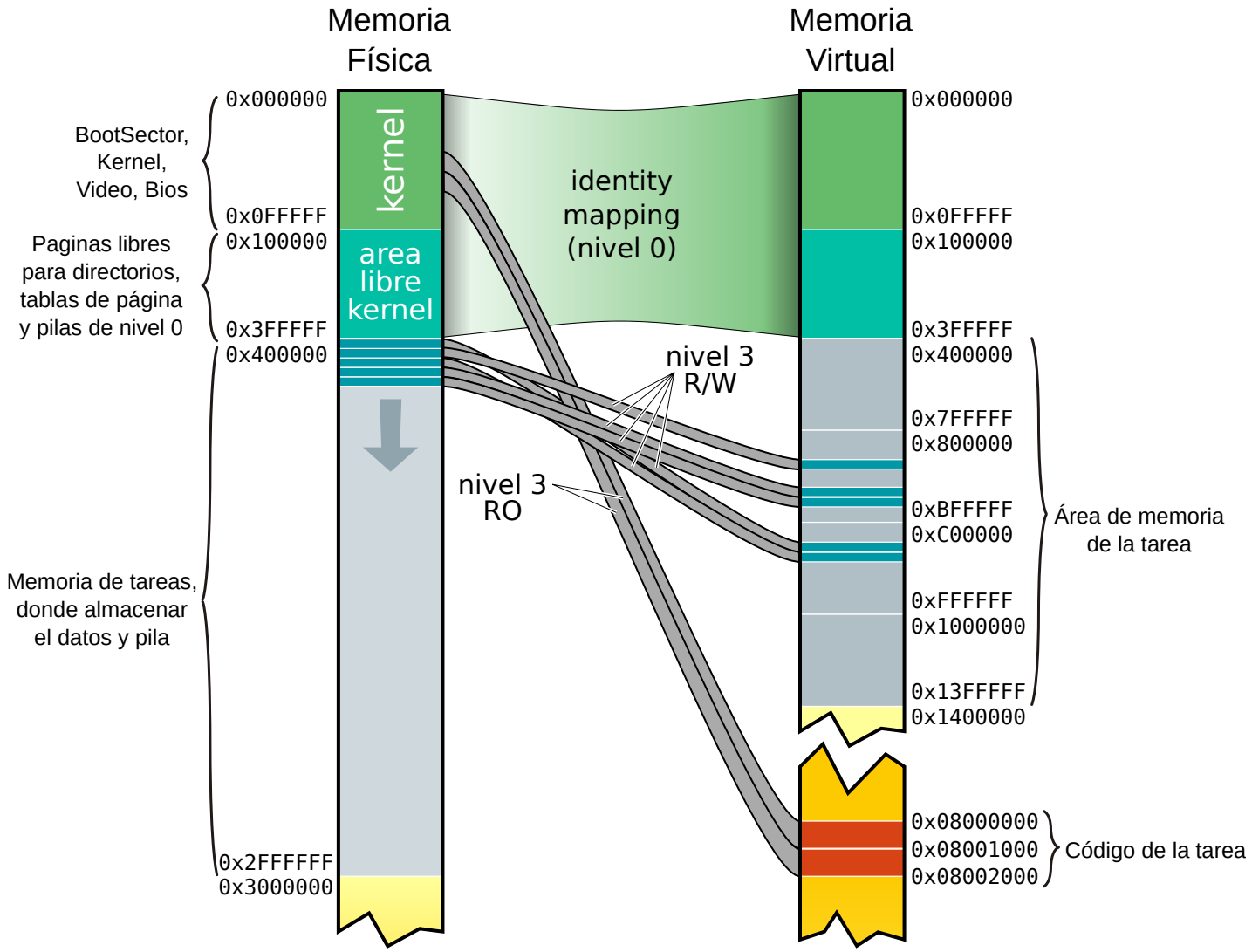


Figura 3: Mapa de memoria de la tarea

La memoria virtual de cada una de las tareas tiene mapeado el *kernel*, *área libre kernel* y *área libre tareas* con *identity mapping* en nivel 0.

El código de las tareas Lemming se encontrará a partir de la dirección virtual *0x08000000* y será mapeado como sólo lectura de nivel 3 a la dirección física correspondiente para el pueblo del que ese Lemming se origina. El stack será mapeado en la página siguiente, con permisos de lectura y escritura. La página física debe obtenerse del *área libre tareas*.

Por último, el sistema les provee a los Lemmings de un mismo pueblo un método de comunicación mediante páginas compartidas, mapeadas bajo demanda. El rango virtual 0x400000 - 0x13FFFFF será reservado para tal fin. Cuando una tarea Lemming es lanzada, tendrá esas páginas desmapeadas. Al acceder a una página dentro de ese rango, si es la primera vez que un Lemming de ese pueblo accede, se mapeará una nueva página física. Si otro Lemming de ese pueblo ya había accedido a esa página, se mapeará la misma página física, permitiendo que los Lemmings se comuniquen por medio de esta memoria compartida.

0.4.3 Scheduler

El sistema va a correr tareas de forma concurrente, durante un tiempo fijo denominado *quantum*. Este será determinado por un *tick* de reloj. Para lograr este comportamiento se va a contar con un *scheduler* minimal que encargará de desalojar una tarea del procesador para intercambiarla por otra en intervalos regulares de tiempo.

Adicionalmente, el objetivo es repartir el tiempo equitativamente entre los dos pueblos. Por esta razón se ejecutará por cada tick de reloj una tarea de un pueblo distinta al del ciclo anterior.

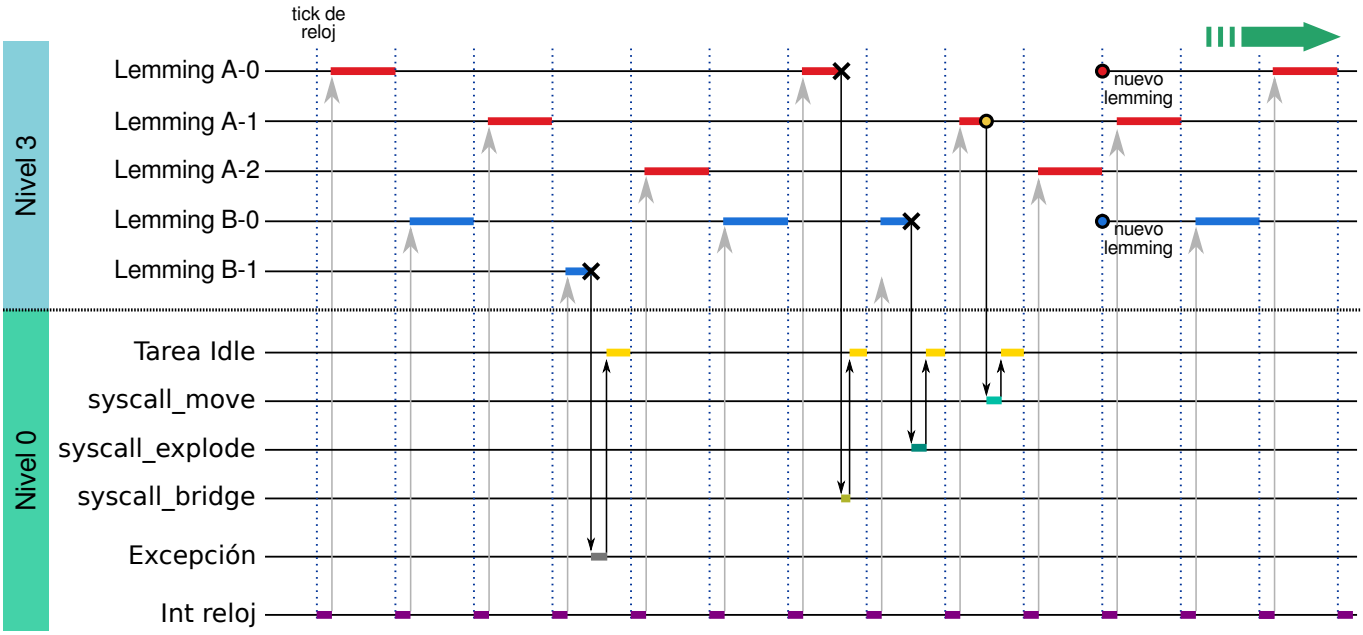


Figura 4: Ejemplo de funcionamiento del *Scheduler*

Las tareas serán ejecutadas una a continuación de la otra, respetando la regla anterior. Este proceso se repetirá indefinidamente. No siempre habrá una tarea Lemming para ejecutar, en ese caso, se ejecutará la tarea Idle.

El orden de ejecución de las tareas puede ser cualquiera, pero se debe garantizar que nunca sean ejecutadas dos tareas seguidas del mismo pueblo Lemming y que todas las tareas sean ejecutadas al menos una vez.

Las tareas pueden generar problemas de cualquier tipo. Por esta razón se debe contar con un mecanismo que permita desalojarlas para que no puedan correr nunca más. Este mecanismo debe poder ser utilizado en cualquier contexto (durante una excepción, un acceso inválido a memoria, un error de protección general, etc.) o durante una interrupción ocasionada por haber llamado de forma incorrecta a un servicio. Cualquier acción que realice una tarea de forma incorrecta será penada con el desalojo de dicha tarea del sistema (esto en el contexto del juego equivale a que la tarea muera).

Un punto fundamental en el diseño del *scheduler* es que debe proveer una funcionalidad para intercambiar cualquier tarea por la tarea `Idle`. Este mecanismo será utilizado al momento de matar a una tarea, ya que la tarea `Idle` será la encargada de completar el *quantum* actual. La tarea `Idle` se ejecutará por el resto del *quantum* de la tarea desalojada, hasta que nuevamente se realice un intercambio de tareas por la próxima que corresponda. Cuando el sistema decida que debe crear una tarea Lemming, al finalizar el proceso de creación, se deberá pasar a la tarea Idle por lo que resta del *quantum*.

En la figura 4 se presenta ... Se puede observar como las tareas llaman a distintos servicios durante su tiempo en ejecución, y como son desalojadas para ejecutar la tarea `Idle` durante el resto del *quantum*.

0.4.4 Modo debug

El sistema deberá responder a una tecla especial en el teclado, la cual activará y desactivará el modo de debugging. La tecla para tal propósito es la "y". En este modo se deberá mostrar en pantalla la primera excepción capturada por el procesador junto con un detalle de todo el estado del procesador como muestra la figura 5. Una vez impresa en pantalla esta excepción, el juego se detendrá hasta presionar nuevamente la tecla "y" que mantendrá el modo debug pero borrará la información presentada en pantalla por la excepción. La forma de detener el juego será instantánea. Al retomar el juego se esperará hasta el próximo ciclo de reloj en el que se decidirá cuál es la próxima tarea a ser ejecutada. Se recomienda hacer una copia de la pantalla antes de mostrar el cartel con la información de la tarea.

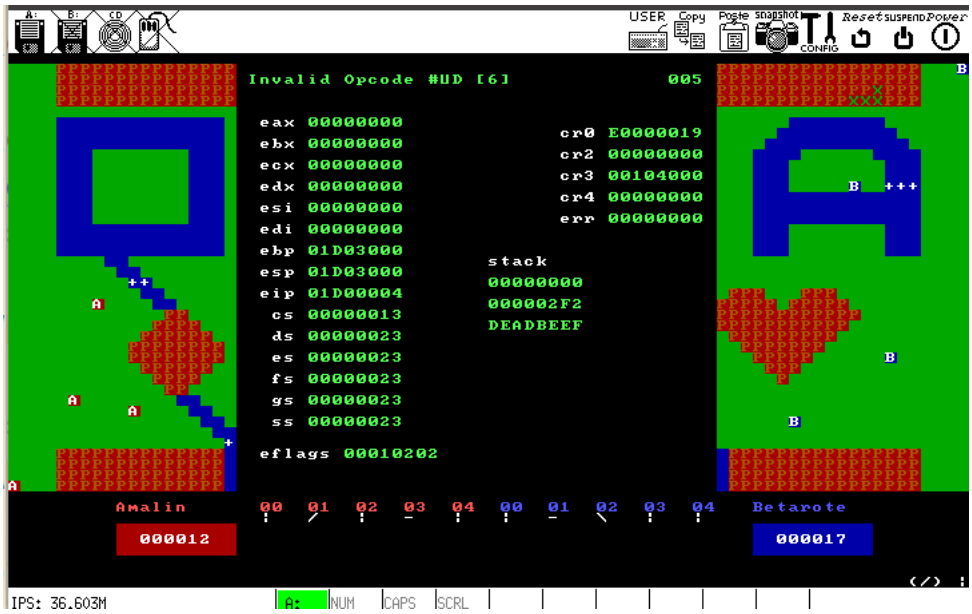


Figura 5: Pantalla de ejemplo de error

0.4.5 Pantalla

La pantalla presentará el *mapa* en forma de mapa de 80×40. En este mapa se indicará el estado actual del terreno, diferenciando los tipos de obstáculo, si hay puentes, o si hay terreno que fue destruido. También se deberán ver las posiciones de todos los Lemmings. Se deberá también indicar la cantidad de Lemmings que fueron creados.

La figura 6 muestra una imagen ejemplo de la pantalla indicando cuáles datos deben presentarse como mínimo. Se recomienda implementar funciones auxiliares que permitan imprimir datos en pantalla de forma cómoda. No es necesario respetar la forma exacta de presentar estos datos en pantalla. Se puede modificar la forma, no así los datos en cuestión.

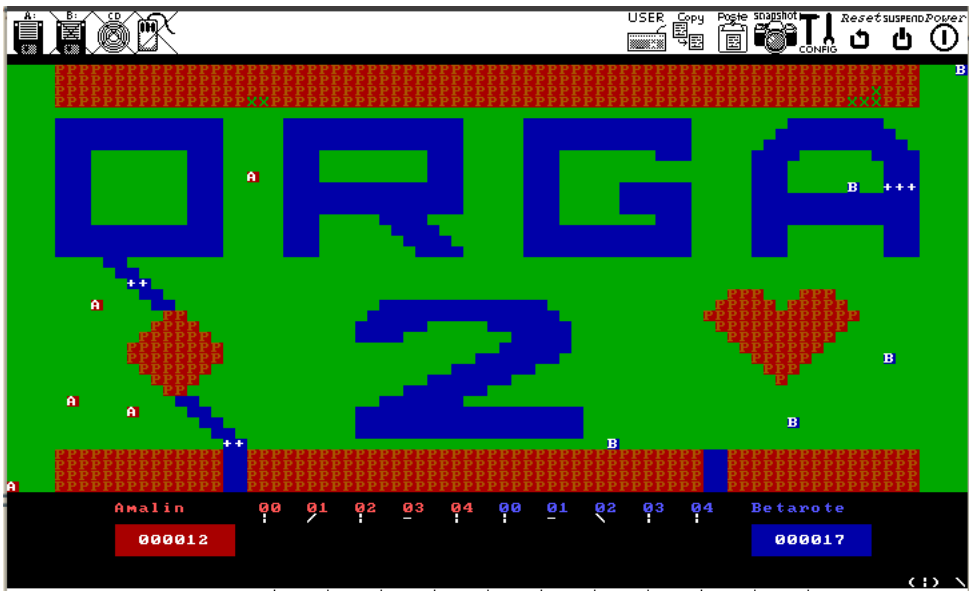


Figura 6: Pantalla de ejemplo

0.5 Ejercicios

Ejercicio 1

a) Completar la Tabla de Descriptores Globales (GDT) con 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 817MiB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 8 posiciones se consideran utilizadas y por ende no deben utilizarlas. El primer índice que deben usar para declarar los segmentos, es el 8 (contando desde cero).

b) Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección 0x25000 (es decir, en la base de la pila).

c) Declarar un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*.

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar [2] el área del mapa con algún

[2] http://wiki.osdev.org/Text_UI

color de fondo, junto con las barras de los jugadores según indica la sección 0.4.5. Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior. Es muy importante tener en cuenta que para los próximos ejercicios se accederá a la memoria de video por medio del segmento de datos.

Nota: La `GDT` es un arreglo de `gdt_entry_t` declarado sólo una vez como `gdt`. El descriptor de la `GDT` en el código se llama `GDT_DESC`.

Con el comando `info gdt i` en bochs pueden ver información sobre la iésima entrada en la gdt.

Ejercicio 2

a) Completar las entradas necesarias en la `IDT` para asociar diferentes rutinas a todas las excepciones del procesador. Cada rutina de excepción debe indicar en pantalla qué problema se produjo e interrumpir la ejecución. Posteriormente se modificarán estas rutinas para que se continúe la ejecución, resolviendo el problema y desalojando a la tarea que lo produjo.

b) Hacer lo necesario para que el procesador utilice la IDT creada anteriormente. Generar una excepción para probarla.

Nota: La `IDT` es un arreglo de `idt_entry_t` declarado sólo una vez como `idt`. El descriptor de la `IDT` en el código se llama `IDT_DESC`. Para inicializar la `IDT` se debe invocar la función `idt_init`.

Con el comando `info idt i` en bochs pueden ver información sobre la iésima entrada en la idt.

Ejercicio 3

a) Completar las entradas necesarias en la `IDT` para asociar una rutina a la interrupción del reloj y otra a la interrupción de teclado. Además crear tres entradas adicionales para las interrupciones de software `88`, `98`, y `108`.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `nextClock`. La misma se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla. La función `nextClock` está definida en `isr.asm`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de `0` a `9`, se presente la misma en la esquina superior derecha de la pantalla.

d) Escribir las rutinas asociadas a las interrupciones `88`, `98`, y `108` para que modifique el valor de `eax` por $0x58$, $0x62$, y $0x6c$, respectivamente. Posteriormente este comportamiento va a ser modificado para atender cada uno de los servicios del sistema.

Ejercicio 4

a) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el *kernel* (`mmu_init_kernel_dir`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x003FFFFF`, como ilustra la figura 3. Además, esta función debe inicializar el directorio de páginas en la dirección `0x25000` y las tablas de páginas según muestra la figura 2.

b) Completar el código necesario para activar paginación.

c) Escribir una rutina que imprima el número de libreta de todos los integrantes del grupo en la pantalla.

Ejercicio 5

a) Escribir una rutina (`mmu_init`) que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre de kernel.

b) Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.

I- `mmu_map_page(uint32_t cr3, uint32_t virtual, uint32_t phy, uint32_t attrs)`
Permite mapear la página física correspondiente a `phy` en la dirección virtual `virtual` utilizando `cr3`.

II- `mmu_unmap_page(uint32_t cr3, uint32_t virtual)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

c) Escribir una rutina (`mmu_init_task_dir`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3.

La rutina debe mapear las páginas de código como solo lectura, a partir de la dirección virtual `0x08000000`, y el stack como lectura-escritura con base en `0x08003000`. La memoria para la pila de la tarea debe salir del area de memoria de las tareas.

d) A modo de prueba, construir un mapa de memoria para tareas e intercambiarlo con el del *kernel*, luego cambiar el color del fondo del primer caracter de la pantalla y volver a la normalidad. Inspeccionar el mapa de memoria con el comando `info tab`. Este item no debe estar implementado en la solución final.

Nota: Por construcción del *kernel*, las direcciones de los mapas de memoria (`page directory` y `page table`) están mapeadas con *identity mapping*.

En las funciones en donde se modifica el directorio o tabla de páginas, se debe llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

Con el comando `page vaddr` en bochs pueden ver información sobre cómo está mapeada la dirección virtual vaddr.

Ejercicio 6

a) Definir las entradas en la `GDT` que considere necesarias para ser usadas como descriptores de `TSS`. Mínimamente, una para ser utilizada por la *tarea inicial* y otra para la tarea Idle.

b) Completar la entrada de la `TSS` de la tarea Idle con la información de la tarea `Idle`. Esta información se encuentra en el archivo `tss.c`. La tarea `Idle` se encuentra en la dirección `0x0001C000`. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con *identity mapping*. Esta tarea ocupa 1 página de 4KiB y debe ser mapeada con *identity mapping*. Además, la misma debe compartir el mismo `CR3` que el *kernel*.

c) Completar la entrada de la `GDT` correspondiente a la *tarea inicial*.

d) Completar la entrada de la `GDT` correspondiente a la tarea `Idle`.

e) Escribir el código necesario para ejecutar la tarea `Idle`, es decir, saltar intercambiando las `TSS`, entre la *tarea inicial* y la tarea `Idle`.

f) Construir una función que complete una `TSS` con los datos correspondientes a una tarea. Esta función será utilizada más adelante para crear una tarea. El código de las tareas se encuentra a partir de la dirección `0x00018000` ocupando dos páginas de 8KiB cada una según indica la figura 2. Para la dirección de la pila se debe usar una página física sin usar, la misma crecerá desde la base de la pila. Para el mapa de memoria se debe construir uno nuevo utilizando la función `mmu_init_task_dir`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva página del área libre de kernel a tal fin.

Nota: En `tss.c` están definidas las `tss` como estructuras `tss_t`. Trabajar en `tss.c` y `kernel.asm`

.

Con el comando `info tss` en bochs pueden ver información sobre la tss actual.

Ejercicio 7

a) Construir una función para inicializar las estructuras de datos del *scheduler*.

b) Crear la función `sched_next_task()` que devuelve el índice en la `GDT` de la próxima tarea a ser ejecutada. Construir la rutina de forma devuelva una tarea por jugador por vez según se explica en la sección 0.4.3.

c) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. Cargar las tareas LemmingA y LemmingB y verificar que se ejecuten (escriben su reloj cuando estan activas, ejecutar system calls de prueba).

El intercambio se realizará según indique la función `sched_nextTask()`.

d) Modificar la rutina de interrupciones 88 para que pase a la tarea idle cuando es invocada. Verificar que la tarea idle se ejecuta (escribe su reloj).

e) Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y ejecuten la próxima.

f) Modificar las rutinas de interrupciones 98 y 108 para que desalojen a la tarea que estaba corriendo y ejecuten la próxima.

g) Implementar el mecanismo de debugging explicado en la sección 0.4.4 que indicará en pantalla la razón del desalojo de una tarea.

Ejercicio 8

h) Inicializar la pantalla del juego.

i) Implementar la lógica de la rutina de interrupciones `move (88)` para mover a un Lemming. Verificar que el Lemming no avance al toparse con obstáculos o los bordes del mapa.

j) Implementar la lógica de la rutina `explode (98)` para explotar. Verificar que las paredes de ladrillos y otros Lemmings se destruyan.

k) Implementar la lógica de la rutina `bridge (108)` para crear un puente. Verificar que ahora otros puedan moverse por donde antes había agua.

l) Implementar la lógica de creación de Lemmings.

m) Modificar la rutina de excepción para *«Fallos de página»* para que asigne las páginas de memoria bajo demanda.

n) Implementar la lógica de finalización del juego.

Ejercicio 9 (optativo)

a) Crear un par de tareas Lemming que respete las restricciones del trabajo práctico, de no hacerlo no podrán ser ejecutados en el sistema implementado por la cátedra.

   Debe cumplir:

   - No ocupar más de 8 KiB
   - Tener como punto de entrada la dirección cero
   - Estar compilado para correr desde la dirección `0x08000000`
   - Utilizar los servicios del sistema correctamente

   Explicar en pocas palabras la estrategia utilizada.

b) Si consideran que su tarea puede hacer algo más que completar el primer ítem de este ejercicio, y se atreven a participar de la competencia de Lemmings, entonces pueden enviar el **binario** de sus tareas a la lista de docentes indicando el nombre de la tarea.

   Se realizará una competencia a fin de cuatrimestre con premios a definir para los primeros puestos.

0.6 Entrega

Este trabajo práctico esta diseñado para ser resuelto de forma gradual. Dentro del archivo `kernel.asm` se encuentran comentarios que muestran las funcionalidades que deben implementarse para resolver cada ejercicio. También deberán completar el resto de los archivos según corresponda.

A diferencia de los trabajos prácticos anteriores, en este trabajo está permitido modificar cualquiera de los archivos proporcionados por la cátedra, o incluso tomar libertades a la hora de implementar la solución, siempre que el producto final resuelva el ejercicio y cumpla con el enunciado. Parte de código con el que trabajarán estará programado en ASM y parte en C, decidir qué se utilizará para desarrollar cada parte de la solución es parte del trabajo.

Se deberá entregar un informe que describa **detalladamente** la implementación de cada uno de los fragmentos de código que fueron construidos para completar el kernel. Se espera que el informe tenga el detalle suficiente como para entender el código implementado en la solución entregada. Considerar al informe como un documento que ayuda al docente corrector en la tarea de entender

su trabajo práctico. En el caso que se requiera código adicional, debe estar descripto en el informe. Cualquier cambio en el código que proporcionamos también deberá estar documentado. Se deberán utilizar tanto pseudocódigos como esquemas gráficos, o cualquier otro recurso pertinente que permita explicar la resolución. Además se deberá entregar en soporte digital el código e informe; incluyendo todos los archivos que hagan falta para compilar y correr el trabajo en Bochs.

La fecha de entrega de este trabajo es **15/06**. Deberá ser entregado a través de un repositorio GIT almacenado en `https://git.exactas.uba.ar` respetando el protocolo enviado por mail y publicado en la página web de la materia. El informe de este trabajo debe ser incluido dentro del repositorio en formato `PDF`.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.