Trabajo Práctico III

System Programming

Organización del computador II

Primer cuatrimestre de 2021

Integrante	L.U.	Correo electrónico
Francisco Herrero	136/19	herrerofranciscojose@gmail.com
Carla de Erausquin	126/18	deercarla@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

Resumen

En este último trabajo práctico de la materia nos encontramos involucrados en la implementación de un muy pequeño Kernel de 32 bits. Para ello contamos con una aproximación por parte de la cátedra en la cuál pudimos tener una noción de la arquitectura de nuestro futuro Kernel.

Índice

1. Introducción

El objetivo de este Trabajo Práctico es tener un primer encuentro con los conceptos de system programming vistos en la última parte de la materia. Para eso, mediante el emulador bochs simularemos un sistema operativo de 32 bits y sobre el mismo trabajaremos a lo largo de 8 ejercicios para poder entender las funcionalidades más importantes de estos sistemas y al final, tener un juego funcional basado en el videojuego "Lemmings". Los temas más importantes a tratar en estos puntos son: segmentos, paginación, interrupciones, tareas, protección y scheduler.

2. Desarrollo

2.1 Ejercicio 1:

GDT - Segmentos I

Se completó la tabla de Descriptores Globales (GDT) dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3 los cuales direccionaron los primeros 817MiB de memoria. Teniendo en cuenta la estructura de un segmento:

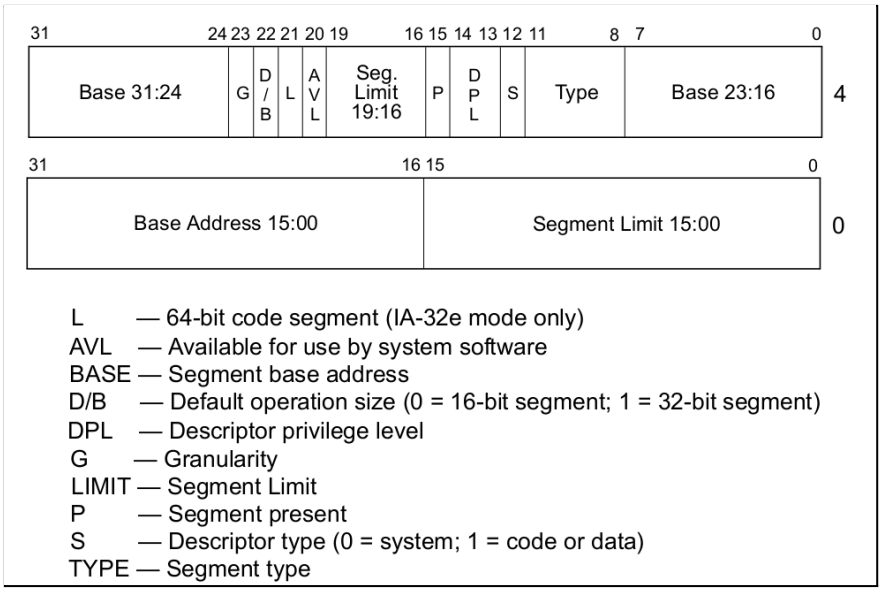


Figura 0: Estructura de un segmento

Se fue completando la GDT respetando los tipos de cada segmento pedido, por ejemplo:

```
[GDT_IDX_C_0] =
     {
          .limit_15_0 = 0x1DDA,
          .limit_19_16 = 0x03,
          .base_15_0 = 0x0000,
          .base_23_16 = 0x00,
          .base_31_24 = 0x00,
          .type = 0xA,
          .s = 0x01,
          .dpl = 0x00,
          .p = 0x01,
          .avl = 0x0,
          .l = 0x0,
          .db = 0x1,
          .g = 0x01,
     }
```

Figura 1: Segmento de GDT correspondiente a código de nivel 0 de lectura y ejecución.

Al tener que direccionar los primeros 817MiB, la base valdra 0, y el límite será tomando a páginas de 4KiB, con la granularidad encendida ya que, cada segmento sólo puede direccionar hasta 1MiB, en caso de granularidad apagada. El tipo se definió según lectura o ejecución según cada segmento, lectura y ejecución (`0xA`) en el caso de la *Figura 1*. Todo segmento que sea de código o datos, tendrá el bit de sistema encendido, luego el bit de default/big estará encendido, ya que estamos trabajando en un sistemas de 32bits, en consecuencia el bit de long valdrá cero y el bit de presente estará encendido Finalmente, según el nivel de privilegio especificado para cada uno de los segmentos (0 ó 3), se completó el *dpl.*

Modo Protegido

Para pasar a modo protegido tuvimos que seguir los siguientes pasos:

- Habilitamos A20 con los siguientes calls:

```
call A20_disable
call A20_check
call A20_enable
call A20_check
```

- Luego cargamos la gdt con el comando lgdt:

```
lgdt [GDT_DESC]
```

- Después de esto seteamos el bit de PE en 1 en el CR0 para habilitar modo protegido. Es necesario pasar el CR0 a EAX ya que no se puede hacer operaciones con el mismo:

```
mov eax, cr0
or  eax, 0x1
mov cr0, eax
```

- Saltamos a modo protegido utilizando el selector de segmento de código de nivel 0:

```
jmp CS_RING_0:modo_protegido
```

- Establecer selectores de segmentos:

```
mov ax, DS_RING_0
mov ds, ax
mov es, ax
mov fs, ax
mov gs, ax
mov ss, ax
```

- Finalmente seteamos la pila del kernel en la dirección 25000:

```
mov ebp, EBP_INIT
mov esp, ESP_INIT
```

GDT - Kernel

Aquí debemos llenar un nuevo descriptor de segmento, en este caso para la memoria de video. Este es un segmento de datos comienza en la dirección 0xB8000 y que ocupa 8000 bytes. Este número se consigue de la siguiente manera: La pantalla es de 80 × 50 posiciones, por lo cual tenemos 4000 posiciones. Además cada una de estas posiciones ocupa 2 bytes, ya que uno de los mismos se utiliza para el caracter que se escribirá en esa posición y el restante se utiliza para setear los colores, tanto del fondo de esa posición como del caracter en sí mismo. De ahí es que obtenemos el valor 8000. Teniendo esto en cuenta rellenamos el segmento destinado a la pantalla siguiendo la lógica anterior como se muestra en la *Figura 2*.

```
[GDT_IDX_V_0] =
   {
         .limit_15_0 = 0x1F3F,
         .limit_19_16 = 0x00,
         .base_15_0 = 0x8000,
         .base_23_16 = 0x0B,
         .base_31_24 = 0x00,
         .type = 0x2,
         .s = 0x01,
         .dpl = 0x0,
         .p = 0x01,
         .avl = 0x0,
         .l = 0x0,
         .db = 0x1,
         .g = 0x00,
   },
```

Figura 2: Segmento de GDT correspondiente que describe el área de la pantalla en memoria que pueda ser utilizado sólo por el kernel.

Pintado de pantalla

Para ello recorrimos el área de video de a pares de bytes, definiendo el valor de carácter como nulo y sus atributos del mismo valor. Para esto cargamos en un registro de segmento el selector del área de video y accedimos mediante la estructura `[selector:offset]`.

2.2 Ejercicio 2:

IDT - Inicialización

En este ejercicio se nos propuso comenzar a manejar las interrupciones genéricas. Para ello, tuvimos que declarar las entradas necesarias en la IDT como se muestra en la siguiente figura:

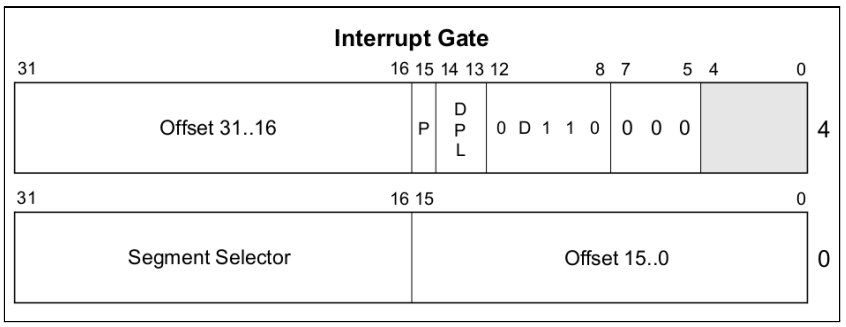


Figura 2: Interrupt Gate struct

Por lo cual, debimos modificar una macro que nos permitió generar dichas entradas a cambio de establecer los valores correctos.

Se definieron los valores en la IDT para las interrupciones desde la 0 a la 19. El selector de segmento donde se encuentra el *handler* de interrupción será el de código de nivel 0, donde se resolvieron las interrupciones, sus atributos son los que definen el *type interrupt gate* y su *dpl* valdrá cero, para finalizar el bit de presente valdrá uno.

Dentro de nuestro Kernel, nos encontramos con la necesidad de agregar varios llamados a distintas funciones e instrucciones para poder, finalmente, inicializar esta nueva estructura y cargarla:

```
;Inicializar la IDT
call idt_init
;Cargar IDT
lidt [IDT_DESC]
```

Figura 4: Interrupt Gate struct

Para probar su correcto funcionamiento, luego de cargarla generamos interrupciones desde kenerl.asm verificando el contexto del sistema operativo chequeando que este atendiendo la interrupción.

2.3 Ejercicio 3:

Se completó la IDT con las entradas necesarias para asociar una rutina a la interrupción del reloj y otra a la interrupción de teclado. Además se crearon tres entradas adicionales para las interrupciones de software 88, 98, y 108.

Dado que las interrupciones deben ser transparentes al momento de comenzar a atender la interrupción, preservamos el contexto de ejecución mediante la instrucción `pushad`, los registros de propósito general. Justo antes de retornar con `iret`, que reactivará las interrupciones, restauraremos el contexto con `popad`.

IDT - Interrupción del reloj I

Para esta interrupción originalmente se pidió que se indicará que la interrupción fue atendida y que se imprimiera el reloj del sistema. Esto se realizó con las funciones `pic_finish1` la cual indica que la interrupción fue atendida y `nextClock` la cual imprime el reloj en pantalla. Ambas fueron implementadas por la cátedra.

```
_isr32:

   pushad

   call pic_finish1
   call nextClock

   popad
   iret
```

Figura 5: Ejemplo de interrupción del reloj

IDT - Interrupción del teclado I

A la hora de implementar la interrupción correspondiente al teclado, se implementó la función `print_scanCode` la cual imprime en pantalla el *ScanCode* de la tecla previamente capturada en la tercera línea de la *Figura 6,* implementada en C. Finalmente, se indica que la interrupción fue atendida con `pic_finish1`.

```
_isr33:

   pushad

   in al, 0x60
   push eax

   call print_scanCode
   add esp, 4

   call pic_finish1

   popad
   iret
```

Figura 6: Ejemplo de interrupción del teclado

IDT - Interrupciones varias I

Para este conjunto de tres interrupciones se pidió, en principio, que únicamente modifiquen el valor de `eax` por otro dado en el enunciado. Para ello se siguió la estructura mostrada en la *Figura 7*:

```
_isr108:
  pushad

   mov dword [esp+7], 0x6c

   popad
   iret
```

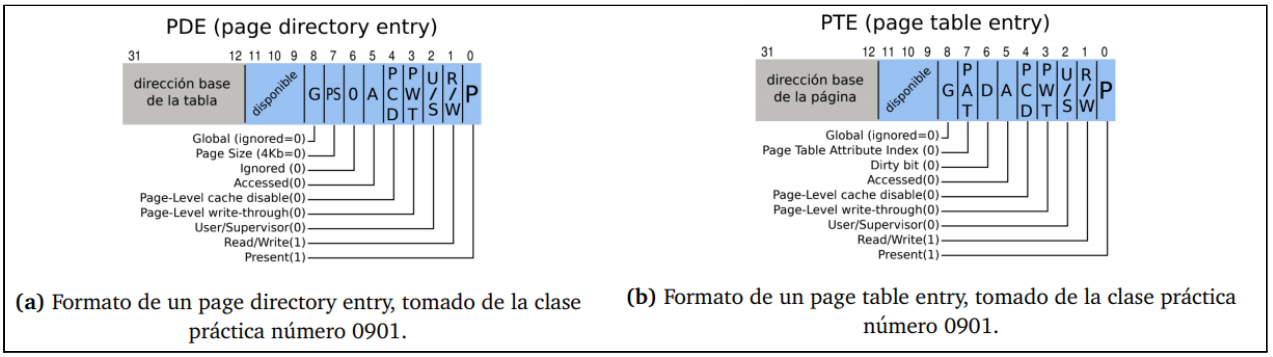
Figura 7: Interrupción 108 original

Vale aclarar que los atributos de las interrupciones 88, 98 y 108 se mantienen a excepción del *dpl* ya que esto nos permite que, si bien se puedan generar estas interrupciones de *software,* no sean ellas mismas las que la resuelvan.

2.4 Ejercicio 4:

Identity mapping

Para resolver este ejercicio se completaron entradas del *page directory* y del *page table* siguiendo las estructuras ilustradas en las a continuación:



(a) Formato de un page directory entry, tomado de la clase práctica número 0901.

(b) Formato de un page table entry, tomado de la clase práctica número 0901.

Figura 8: Estructura de una PDE y PTE

Luego, se escribieron las rutinas encargadas de inicializar el directorio y tablas de páginas para el kernel con la función `mmu_init_kernel_dir` como muestra la *Figura 9*:

```
paddr_t mmu_init_kernel_dir(void) {
     page_directory_entry *pd_k = (page_directory_entry *) KERNEL_PAGE_DIR;

     page_table_entry *pt_0 = (page_table_entry *) KERNEL_PAGE_TABLE_0;
            for (int i = 0; i < 1024; i++) {
                  pd_k[i] = (page_directory_entry){0};
                  pt_0[i]= (page_table_entry){0};
            }
```

```
    mmu_init_page_directory_entry(pd_k, pt_0, 0, MMU_P | MMU_S | MMU_W);
    void mmu_init_page_table_entry(page_table_entry *pt, vaddr_t virt,paddr_t phy,
uint32_t attrs, uint32_t pages)
    mmu_init_page_table_entry(pt_0, 0, 0, MMU_P | MMU_S | MMU_W, 1024);

      return (paddr_t) pd_k;
     }
```

Figura 9: Función `mmu_init_kernel_dir`, la cual retorna el page directory para interpretarlo como CR3

Esta función además inicializa el directorio de páginas en la dirección 0x25000 y las tablas de páginas.

Por otro lado, se generó un directorio de páginas que *mapeara*, usando *identity mapping,* las direcciones las direcciones 0x00000000 a 0x003FFFFF siguiendo la estructura ilustrada en la la *Figura 10* para la traducción con páginas de 4 bytes.

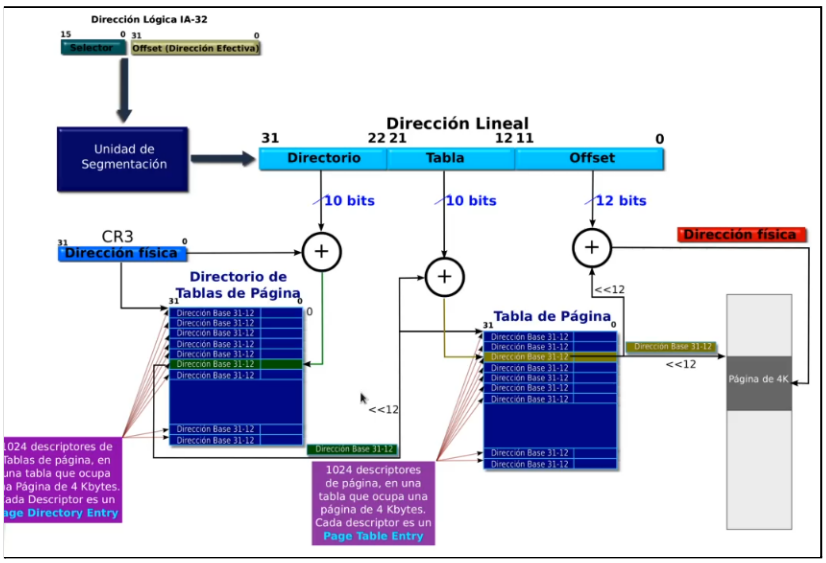


Figura 10: Traducción con páginas de 4 bytes

Activación de paginación

Para activar paginación debemos encender el bit PG en el control register CR0, como muestra la *Figura 11:*

```
mov eax, cr0
mov edx, FLAG_PG
or  eax, edx
mov cr0, eax
```

Figura 11: Código para la activación de paginación

Cabe destacar que el sentido del uso de paginación es dar dinamismo al manejo de la memoria física disponible, y como se puede concluir, esto nos permite manejar de una manera eficiente más memoria de la que físicamente disponemos. En el cuadro anterior se puede apreciar el funcionamiento de la traducción que realiza el sistema de paginación para obtener de una dirección lineal, la dirección efectiva o física.

2.5 Ejercicio 5:

Administración de memoria

La rutina `mmu_init` inicializa las estructuras para el manejador de memoria, desde el comienzo del área de paginas en el kernel directorios, tablas y pilas de nivel 0, hasta el comienzo del área de páginas para el área compartida y pila de usuarios además de inicializar dos arreglos que sirven para tener un control de las páginas compartidas de cada equipo.

```
void mmu_init(void) {
   next_page_kernel = 0x100000;
   next_page_user = 0x2400000;
   virt_amalin_compartida = (uint8_t(*)[4096]) mmu_next_free_kernel_page();
   virt_betarote_compartida = (uint8_t(*)[4096]) mmu_next_free_kernel_page();
  }
```

Figura 12: Código de `mmu_init`

Mapeo y des-mapeo

La función `mmu_map_page` recibe entre sus parámetros: un cr3, una dirección virtual, una dirección física y un conjunto de atributos. Al momento de generar el mapeo, sólo tendrá en consideración el estado de la entrada del directorio. Si esta no se encuentra presente, entonces se pide una página al área libre de kernel y se inicializa en ceros para luego inicializar la entrada correspondiente en el directorio con `mmu_init_page_directory_entry()` (Anexo: FIgura 15). En cualquiera de los dos casos, se llama a `mmu_init_page_table_entry()` (Anexo: FIgura 16) que se encarga de inicializar la entrada en la tabla. Para finalizar, se ejecuta la función `tlbflush`(), para invalidar todas las traducciones del caché de traducciones, por si sobre escribio algún mapeo al momento de llamar esta función.

```
void mmu_map_page(uint32_t cr3, vaddr_t virt, paddr_t phy, uint32_t attrs) {
      page_directory_entry *pd = (page_directory_entry*) (cr3 & RM_ATTRS);
```

```
       vaddr_t directory_index = virt>>22; //dirct_indx
       if(pd[directory_index].present!=1){
       page_table_entry *new_pt = (page_table_entry *) mmu_next_free_kernel_page();
          for(int i = 0; i < 1024; i++){
            new_pt[i] = (page_table_entry){0};
            }
            mmu_init_page_directory_entry(pd, new_pt, virt, attrs);
        }
      page_table_entry *pt = (page_table_entry*)(pd[directory_index].page_table_base
<<12);
        mmu_init_page_table_entry(pt, virt, phy, attrs, 1);
        tlbflush();
      }
```

Figura 13: Función `mmu_map_page`, encargada de mapear una página

En el caso de `mmu_unmap_page` se recorre toda la estructura, desde el directorio hasta la entrada en la tabla de páginas, y se actualizará su valor de presente en cero.

```
paddr_t mmu_unmap_page(uint32_t cr3, vaddr_t virt)
     paddr_t r;
     page_directory_entry *direct_base = (page_directory_entry*) (cr3 & RM_ATTRS);
     vaddr_t direct_indx = virt>>22;
     page_table_entry*table_base = (page_table_entry*)
     (direct_base[direct_indx].page_table_base << 12);
     uint32_t table_indx = ((uint32_t) (virt>>12)) & 0x3FF; //table_indx
     r = (paddr_t) (table_base[table_indx].physical_address_base << 12);
     table_base[table_indx].present = 0;
     tlbflush();
     return r;
      }
```

Figura 14: Función `mmu_unmap_page`, encargada de des-mapear una página

El directorio de una tarea se compone de tres mapeos en común para todas ellas, excepto la idle y la inicial. Inicialmente cada tarea contará con un mapeo del área de kernel, el área de código, a partir de la dirección 0x8000000 hasta 0x8002000, y el área de pila con su base en la dirección 0x8003000.

La función comienza generando las estructuras necesarias para resolver el mecanismo de paginación, reservando e inicializando el page directory y la page table del área de kernel.

Luego mapeamos el área de código de la tareas, que serian dos entradas en la page table.

Para finalizar se mapea el area compartida por los lemming de un mismo equipo, dejando esas entradas, tanto de directorio como de table con el bit de presente en 0.

```
paddr_t mmu_init_task_dir(paddr_t phy, uint8_t team) {
    page_directory_entry *pd_task = (page_directory_entry *)
mmu_next_free_kernel_page();
    page_table_entry *pt_0 = (page_table_entry *) mmu_next_free_kernel_page();

    for (int i = 0; i < 1024; i++) {
       pd_task[i] = (page_directory_entry){0};
       pt_0[i]= (page_table_entry){0};
    }

    mmu_init_page_directory_entry(pd_task, pt_0, 0, MMU_P | MMU_S | MMU_W);
    mmu_init_page_table_entry(pt_0, 0, 0, MMU_P | MMU_S | MMU_W, 1024);

    page_table_entry *pt_cdst = (page_table_entry*) mmu_next_free_kernel_page();

    mmu_init_page_directory_entry(pd_task, pt_cdst, TASK_CODE_VIRTUAL, MMU_P | MMU_U |
MMU_W);
    mmu_init_page_table_entry(pt_cdst, TASK_CODE_VIRTUAL, phy, MMU_P | MMU_U | MMU_R, 2);

    mmu_init_page_table_entry(pt_cdst, TASK_STACK_BASE-0x1000, mmu_next_free_user_page(),
MMU_P | MMU_U | MMU_W, 1);

    paddr_t phy_shared;
    if(team == AMALIN){
       phy_shared = (paddr_t) (0x400000);
    }else{
       phy_shared = (paddr_t) (0x1400000);
    }
    for(int i = 0; i < 4; i++){
       page_table_entry *pt_shared = (page_table_entry *) mmu_next_free_kernel_page();
       phy_shared += (0x400000 * i);
       vaddr_t virt = INICIO_AREA_COMPARTIDA+0x400000 * i;
       mmu_init_page_directory_entry(pd_task, pt_shared, virt, MMU_U | MMU_W);
       mmu_init_page_table_entry(pt_shared, virt, phy_shared, MMU_U | MMU_W, 1024);
    }

    return (paddr_t) pd_task;
}
```

Figura 17: Función `mmu_init_task_dir`, encargada de pegarse un tiro

2.6 Ejercicio 6:

GDT - Segmentos II

Similar a lo enunciado en la sección *2.1 GDT - Segmentos I,* se completó la tabla GDT, ahora con segmentos destinados que luego fueron empleados como descriptores de la TSS. Se necesitaron en total doce nuevas entradas: una para la tarea *idle*, otra para la tarea inicial, cinco para los jugadores del equipo Amalín y otras cinco para el equipo Betarote.

Siendo que los 12 segmentos mencionados se diseñaron como descriptores de la TSS, el nivel de privilegio (*dpl*) de todos ellos es 0. En cuanto al límite, se definió en 67 bytes (como mínimo) por tratarse de una TSS. El resto de los valores se completaron según lo indicado en la *Figura 18:*

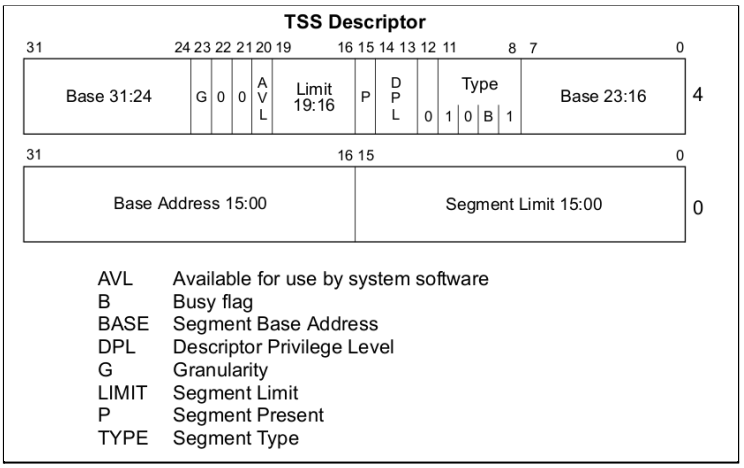


Figura 18: Descriptor de segmento para una TSS

Tarea Inicial

La TSS de la tarea inicial sólo será utilizada en el primer *task-switch*, por lo tanto no debe ser inicializada, basta con crear un segmento de código válido para completar los requerimientos del mecanismo y actualizar el descriptor de la `tss_init` con la dirección base de la tss_init.

```
void tss_initial_init(void){
   gdt[GDT_TSS_INIT].base_15_0  = (uint32_t )(&tss_initial) & 0xFFFF;
   gdt[GDT_TSS_INIT].base_23_16 = ((uint32_t )(&tss_initial) >> 16) & 0xFF;
   gdt[GDT_TSS_INIT].base_31_24 = ((uint32_t )(&tss_initial) >> 24) & 0xFF;
}
```

Figura 19: Código de `tss_initial_init`, encargada de actualizar el descriptor de la `tss_init`

```
[GDT_TSS_INIT] =
{
   .limit_15_0 = 0x0067,
   .limit_19_16 = 0x0,
   .base_15_0 = 0x0000,
   .base_23_16 = 0x00,
   .base_31_24 = 0x00,
   .type = 0x9,
   .s = 0x00,
   .dpl = 0x0,
   .p = 0x01,
   .avl = 0x0,
   .l = 0x0,
   .db = 0x0,
   .g = 0x00,
},
```

Figura 20: Segmento correspondiente a la tarea inicial

Tarea Idle

La tarea idle comparte cr3 con el kernel, por lo tanto la mayoría de las asignaciones de registros de segmento se justifican siendo los de nivel cero. El eip valdrá 0x1C000, la dirección donde se encuentra el código de la tarea Idle dentro del kernel y los eflags tendrán las interrupciones encendidas.

```
void tss_idle_init(void){
   tss_idle.eip = TASK_IDLE_START;
   tss_idle.cr3 = KERNEL_PAGE_DIR;
   tss_idle.esp = KERNEL_STACK;
   tss_idle.ebp = KERNEL_STACK;
   tss_idle.cs  = CS_RING_0;
   tss_idle.ds  = DS_RING_0;
   tss_idle.es  = DS_RING_0;
   tss_idle.fs  = DS_RING_0;
   tss_idle.gs  = DS_RING_0;
   tss_idle.ss  = DS_RING_0;
   tss_idle.eflags = EFLAGS_I;

   gdt[GDT_TSS_IDLE].base_15_0  = (uint32_t )(&tss_idle) & 0xFFFF;
   gdt[GDT_TSS_IDLE].base_23_16 = ((uint32_t )(&tss_idle) >> 16) & 0xFF;
   gdt[GDT_TSS_IDLE].base_31_24 = ((uint32_t )(&tss_idle) >> 24) & 0xFF;
}
```

Figura 21: Inicialización de la TSS idle

```
[GDT_TSS_IDLE] =
{
   .limit_15_0 = 0x0067,
   .limit_19_16 = 0x0,
   .base_15_0 = 0x0000,
   .base_23_16 = 0x00,
   .base_31_24 = 0x00,
   .type = 0x9,.s = 0x00,
   .dpl = 0x0,.p = 0x01,
   .avl = 0x0,.l = 0x0,
   .db = 0x0,
   .g = 0x00,
},
```

Figura 22: Segmento correspondiente a la tarea idle

Idle task-switch

Para comenzar a correr la tarea idle, deberemos cargar el registro TR, con el selector de segmento de la TSS de la tarea Init, para luego comenzar el task-switch con la tarea Idle, basándonos en la estructura jmp seg_sel:offset, con el selector de la TSS de la tarea Idle.

```
mov ax, TSS_INIT
ltr ax

jmp TSS_IDLE:0
```

Figura 23: Código correspondiente al inicio de la tarea idle

Inicializar la TSS de una tarea

El código es similar al que utilizamos para inicializar la TSS de la tarea Idle, por lo tanto basta con cambiar las asignaciones.

El eip es 0x8000000, el cr3 corresponde al directorio de la tarea, creada por `mmu_init_task_dir`(phy_code,team), la base de la pila de nivel 3 y el tope valen 0x80003000, todos los registros de segmento excepto el de código, estarán inicializado con el selector del segmento de datos de nivel 3, y el de código, con el segmento de código de nivel 3. Para finalizar, en los eflags estarán habilitadas las interrupciones y la pila de nivel cero, estará asignada al segmento de datos de nivel cero, y su esp comenzará después del último byte direccionable de la página que se pidió al área de kernel.

2.7 Ejercicio 7:

Inicializar scheduler

El *scheduler* cuenta con algunas estructuras que se inicializan con `sched_init`(), tales como un arreglo de 10 *segsel*, que contiene los selectores de TSS de ambos equipos de leming agrupadas entre sí en orden ascendente. Las primeras cinco TSS son del equipo Amalin y las restantes del equipo Betarote. Otro par de arreglos (uno para Amalin y otro para Betarote), que sirven para llevar el control de qué tareas se encuentran vivas de un equipo en específico, también cuenta con tres variables de tipo entero que se utilizan para llevar la cuenta de la cantidad de lemmings vivos para alguno de los equipos y un contador de ciclos para saber cuándo se debe crear o desalojar un lemming. Para finalizar, contamos con un último par de arreglos que se utilizan para saber cuántos turnos lleva en vida un lemming en específico de alguno de los equipos. Todas estas estructuras se inicializan en cero a excepción del arreglo de 10 TSS, que se cargará con los selectores de segmento correspondiente a cada Lemming.

Siguiente tarea scheduler

La función `sched_next_task`() hará uso de variables globales tales como `last_player`, que contiene el último equipo que corrió, otras dos variables que contienen la última tarea ejecutada de cada equipo y de los arreglos que dado un índice de tarea y su equipo, podemos saber si esta está viva y debe ser ejecutada.

El cuerpo de la función se abre en dos ramas según si el último equipo en correr fue el Amalin o el Betarote. Para hacer más amena la explicación tomaré uno de los dos casos pero el otro es análogo.

La función verifica el último equipo en correr y actualizará dicha variable con el equipo contrario, avanza a la próxima tarea del equipo actual y verifica que al hacerlo, si ya se ejecutó la última tarea en orden, entonces se volverán a ejecutar desde el comienzo. Para terminar se verifica si dicha tarea se encuentra viva, en caso de estarlo se actualiza la cantidad de turnos que lleva viva el nuevo lemming y se retorna su selector de segmento en la GDT. En caso de no estar vivas, se retorna el de la tarea *Idle*.

```
uint16_t sched_next_task(void) {
  if(last_player == AMALIN){
      last_player = BETAROTE;
      last_betarote_task++;
      if(!(last_betarote_task < PLAYER_TASKS)){
      last_betarote_task = 0;
      }
      if(tareas_vivas_betarote[last_betarote_task] == 0){
      return TSS_IDLE;
      }else{
      turn_betarote_created[last_betarote_task]++;
      return tareas_betarote[last_betarote_task];
      }
  }else{....}
}
```

Figura 24: Función que retorna el segsel de la próxima tarea a ejecutar.

Cambio de tareas por ciclo

Para realizar el intercambio de tareas por cada ciclo de clock modificamos la rutina de atención de reloj, llamamos a `sched_next_task`(), para luego verificar que el selector que obtenemos no era el mismo de la tarea que se está ejecutando actualmente, para evitar un error de protección general, luego en caso de no ser el mismo, hacemos uso de la estructura definida para hacer jmp far.

```
;; Rutina de atención del RELOJ
_isr32:
      pushad
      call pic_finish1   ;Indica que la interrupción fue atendida

      call next_clock    ;Imprimir el reloj del sistema

      call sched_next_task
      str bx
      cmp ax, bx
      jz .fin
      mov word [sched_task_selector], ax
      jmp far [sched_task_offset]

      .fin:
      popad
      iret
```

Figura 25: Función que retorna el segsel de la próxima tarea a ejecutar.

Modificar interrupción 88

```
;; Rutinas de atención de las SYSCALLS
_isr88:
      pushad
      jmp TSS_IDLE:0
```

```
popad
iret
```

Figura 26: Función que retorna el segsel de la próxima tarea a ejecutar.

Rutina excepción e interrupciones 98 y 108

Llamamos a la función `sched_kill_task`() (*Figura 28*) encargada de matar a la tarea actual para luego saltar a la tarea *idle* como se muestra a continuación.

```
_isr%1:
      call sched_kill_task
      jmp TSS_IDLE:0x0
```

Figura 27: Función que retorna el segsel de la próxima tarea a ejecutar.

En cuanto a la función mencionada anteriormente, esta se encarga de "matar" la tarea actual modificando el arreglo `tareas_vivas_(x)`, donde la "x" representa a alguno de los dos equipos en cuestión de forma que en el índice de la última tarea ejecutada del equipo "x" sea 0. Luego la intercambiará con la tarea *Idle* hasta que se complete el *quantum* actual.

Para las interrupciones 108 y 98 se siguió un razonamiento análogo al mencionado anteriormente para que desalojen a la tarea que estaba corriendo y ejecuten la próxima.

2.8 Ejercicio 8:

Inicialización de pantalla

Separamos la inicialización de la pantalla en tres pasos, primero cargamos el mapa a través de `load_map`()(*Figura 30*). Esta función se encarga de cargar el mapa del juego recorriendo por filas y actualizando un mapa de respaldo llamado `mapa_mod` el cual será el lugar donde se guardan las modificaciones del mapa original según la partida. Luego pintaremos los puntajes de cada uno con `print_score`() (*Figura 31),* y pintaremos los equipos con sus respectivos relojes con `print_teams`() *(Figura 32)*.

```
void game_init(void) {
      load_map();
      print_scores();
      print_teams();
}
```

Figura 29: Función encargada de inicializar la pantalla del juego.

Interrupción "move" (88)

Para la lógica del desplazamiento de lemmings, volvimos a modificar la rutina de atención agregando un llamado a la función `move_main`. Para ello capturamos los parámetros recibidos en `eax` a través de la pila. Para finalizar retornamos por `eax` el resultado de la operación modificando el espacio de memoria que le corresponde al registro al momento de hacer `popad`.

```
_isr88:
      pushad
      push eax
      call move_main
      mov [esp+8*4], eax
      ;esp+7*4 = eax_ret, pero se pusheo el parámetro,
      ; entonces esp+8*4 = eax_ret
      add esp, 4
      jmp TSS_IDLE:0
      popad
      iret
```

Figura 33: Atención a la interrupción del Syscall move

La función `move_main`(...)(*Figura 34*) verifica si el movimiento es válido y en consecuencia actualiza la posición del lemming. Para ratificar que el movimiento sea válido, se llama a la función `move_valido`(...)(*Figura 35*) la cual según el caso (arriba, abajo, izquierda y derecha) llama a su vez a funciones análogas a la *Figura 36* las cuales se encargan de comprobar que el lemming en cuestión no posea un obstáculo, tal como una pared, un lago u otro lemming, el cual le impida realizar el movimiento.

```
move_result_e move_up(pos_t *pos){
      ca (*screen)[ROW_SIZE] = (ca (*)[ROW_SIZE])VIDEO;
      move_result_e res = SUCCESS;
      if(pos->x == 0){
         res = BORDER;
      }else{
         char simbol = screen[pos->x-1][pos->y].c;
         if(simbol == *"P"){
           res = WALL;
         }else{
            if(simbol == *"L"){
              res = LAKE;
            }else{
              if(simbol == *"A" || simbol == *"B"){
                res = LEMMING;
              }
            }
         }
      }
}
```

Figura 36: Función de movimiento hacia arriba

Interrupción "explode" (98)

Se modificó la interrupción 98 la cual hace dos llamados a función como se puede ver en la *Figura 37,* para luego realizar el salto a la tarea *idle* similar a la interrupción 88:

```
_isr98:
      pushad
      call explode_main
      call sched_kill_task
      jmp TSS_IDLE:0
      popad
      iret
```

Figura 37: Interrupción 98 final, encargada de la lógica de la explosión de un lemming

Similar a *move*, esta interrupción llama también a una función principal la cual llama a su vez a varias auxiliares. La función `explode_main` toma la posición del lemming actual según su etnia y verifica qué hay en su entorno:

```
void explode_main(void){
      int8_t team = sched_actual_team();
      int8_t task = sched_actual_task(team);
      pos_t *pos;
      if(team == AMALIN){
      pos = &pos_A[task];
      }else{
      pos = &pos_B[task];
      }
      load_simbols(pos);
      for(int i = 0; i < 8; i++){
      if(simbols[i] == *"P"){
            explode_wall(pos, i);
      }
      if(simbols[i] == *"A" || simbols[i] == *"B"){
            explode_leming(pos, i);
      }
      }
      print_ground(pos);
      return;
}
```

Figura 38: Función `explode_main`, encargada de controlar la población de lemmings

Independientemente de lo que haya, el lemming explotará pero, de haber algo alrededor de él, la función llamará a `explode_wall()` o `explode_leming()` según corresponda. La primera se encarga sencillamente de eliminar la pared cercana al lemming borrándose del mapa. La segunda frena las tareas de los lemmings cercanos al explosivo.

Finalmente, el lemming en cuestión debe morir (son criaturas sensibles a la pirotecnia). Esto es realizado por la función `sched_kill_task (Figura 28)` la cual se encarga de frenar la tarea del susodicho ya que una vez inmolado, el lemming no puede realizar tarea alguna.

Interrupción "bridge" (108)

La interrupción *bridge* ilustrada en la siguiente figura posee un funcionamiento análogo a la explosión. Se llama a una función principal, `bridge_main` la cual se encarga de construir un puente con un funcionamiento similar a `explode_main` sólo que esta es más sencilla ya que únicamente debe modificar el mapa en la posición de agua sobre la cual se quiere construir el puente.

```
_isr108:
      pushad
      push eax
      call bridge_main
      add esp, 4
      call sched_kill_task
      jmp TSS_IDLE:0
      popad
      iret
```

Figura 39: Interrupción 108 final, encargada de la lógica de la construcción de puentes

Una vez más, el lemming en cuestión debe morir con el llamado a `sched_kill_task` ya que, aparentemente, no sobrevive a ser usado como puente.

Lógica de creación de Lemmings

Para la lógica de creación de lemmings agregamos un llamado en la rutina de Reloj a la función `sched_control_lemmgins`(), que se encarga de verificar y llevar a cabo la creación de lemming y el desalojo según la convención en el inciso 0.4 del enunciado. En caso de que corresponda crear un lemming, se actualizará el estado de la tarea viva en el arreglo del equipo correspondiente, se cargará en pantalla en la base y se aumentará la variable que lleva el contador de lemmings en juego, su contador en el arreglo de turnos vivos para dicho lemming se pondrá en cero, se llamará a la función `tss_task_reset`(), que reiniciara la tss del lemming al que se revive y por último se imprime en pantalla el clock inicial.

```
tareas_vivas_amalin[idx] = 1;
pos_init(&pos, AMALIN, idx);
vivas_amalin++;
turn_amalin_created[idx] = 0;
tss_task_reset(AMALIN, idx);
print("|", 19+idx*4, 45, 0x0F);
```

Figura 40: Código que crea un nuevo lemming

En caso de desalojar al lemming más antiguo, se llamará a la función `sched_kill_old`() se encarga de buscar dentro del arreglo que los turnos y capturar el indice del lemmings con más turnos vivo, cabe aclarar que cuando un lemming muere o es creado, el valor del arreglo en su índice vale cero. En caso de que exista una tarea a ser desalojada, se tendrá en cuenta dos casos, si la tarea que se desaloja es la actual o si es cualquier otra, en ambos casos se hace lo mismo, se llama a la función `sched_explode`(), que tiene el mismo funcionamiento que `sched_kill_task`(), la diferencia que deja fuera de ejecución una tarea en específico, en cualquiera de los dos casos diferencia radica al retornar a la interrupción de reloj, si la tarea actual debía ser desalojada, al retornar de C, se hará el intercambio de tarea con la Idle.

Excepción "fallos de página"

La rutina de fallo de página se modificó en función de satisfacer la necesidad de complementar el funcionamiento del juego, en el cual los equipos cuentan con un área compartida de memoria que debe ser mapeada baja demanda, la cual está pre asignada en el mapa de memoria de cada una de las tareas, y se ira actualizando su estado a medida de que se hagan accesos de algún tipo, concretando el mapeo al momento de activar el bit de presente tanto en el directorio de páginas como en la tabla de páginas, según corresponda.

Para ello hacemos uso de la función implementada en C mmu_map_compartida(), que comienza verificando si la dirección virtual que generó el page fault fue dentro del área compartida, en caso de que no sea así al retornar a la _isr14 se procederá como cualquier otra excepción, matando la tarea actual y haciendo un task switch con la Idle.

Se calcula el índice dentro del directorio de página en función de la dirección virtual que genere la excepción. Luego se obtiene al cr3 de la tarea y se verifica que el bit de presente de la entrada de directorio esté encendida para, de no estarlo, encenderla . A continuación se realiza algo similar para la entrada de tabla, se busca y se enciende su bit de presente, antes de retornar se llama a `tlbflush`().

```
_isr14:
      ;xchg bx, bx
      pushad
      mov eax, cr2
      push eax
      call mmu_map_compartida
      add esp, 4
      cmp eax, effective_map
```

```
      jnz .desalojar

      popad
      add esp, 4  ;Eliminar el error code
      ;   xchg bx, bx
      iret

      .desalojar:
      xchg bx, bx
      call sched_kill_task

      jmp TSS_IDLE:0x0
      jmp $
```

Figura 41: Rutina de atención de page fault

```
uint8_t mmu_map_compartida(vaddr_t virt){
    uint8_t res;
    if(INICIO_AREA_COMPARTIDA<=virt && virt<FIN_AREA_COMPARTIDA){
       uint32_t virt_base = virt & 0xFFFFF000;
       uint32_t virt_idx = (virt_base %0x400000)>>12;
       uint8_t team = sched_actual_team();
       if(team == AMALIN){
             if(*virt_amalin_compartida[virt_idx] == 0){
                   *virt_amalin_compartida[virt_idx] = 1;
             }
             page_directory_entry *pd_task = (page_directory_entry *) tss_task_cr3(team,
sched_actual_task(team));
             vaddr_t directory_index = virt>>22;
             if(pd_task[directory_index].present != 1){
                   pd_task[directory_index].present = 1;
             }
             page_table_entry *pt_task = (page_table_entry *)
(pd_task[directory_index].page_table_base <<12);
             uint32_t table_index = ((uint32_t) (virt>>12)) & 0x3FF; //table_indx
             pt_task[table_index].present = 0x1;
             tlbflush();
       }
       if(team == BETAROTE){
             if(*virt_betarote_compartida[virt_idx] == 0){
                   *virt_betarote_compartida[virt_idx] = 1;
             }
             page_directory_entry *pd_task = (page_directory_entry *) tss_task_cr3(team,
sched_actual_task(team));
             vaddr_t directory_index = virt>>22;
             if(pd_task[directory_index].present != 1){
                   pd_task[directory_index].present = 1;
             }
             page_table_entry *pt_task = (page_table_entry *)
(pd_task[directory_index].page_table_base <<12);
             uint32_t table_index = ((uint32_t) (virt>>12)) & 0x3FF; //table_indx
             pt_task[table_index].present = 0x1;
             tlbflush();
       }
       res = 1;
    }else{
       res = 0;
```

```
    }
    return res;
}
```

Figura 42: Rutina para mapear el área compartida de un equipo

Lógica de finalización de juego

Se modificó la rutina de atención de reloj, agregando un llamado a la función game_cheack(), la cual verifica si en alguna de las columnas borde hay un lemming del equipo que se encuentra del lado opuesto. En caso de ser se ejecuta la función game_end(), que imprimirá un mensaje en pantalla denotando al equipo ganador, y se quedará en un loop infinito (game_end_ASM).

```
void game_cheack(void){
      int8_t res = -1;
      for(int i = 0; i < 5; i++){
      if(clocks_A[i]!=-1 && pos_A[i].y ==79){
            res = AMALIN;
      }
      if(clocks_B[i]!=-1 && pos_B[i].y ==0){
            res = BETAROTE;
      }
      }
      if(res != -1){
      game_end(res);
      }
}
```

Figura 43: Función que verifica la condición de victoria para alguno de los dos equipos

```
void game_end(int8_t team){
      if(team == AMALIN){
      print_A_wins();
      }else{
      print_B_wins();
      }
game_end_ASM();
}
game_end_ASM:
      jmp $
```

Figura 44: Función que imprime en pantalla al equipo ganador y congela la ejecución del juego.

3. Apéndice

```
void  mmu_init_page_directory_entry(page_directory_entry  *pd,  page_table_entry  *pt,
vaddr_t virt, uint32_t attrs){
     vaddr_t directory_index = virt>>22;
     pd[directory_index] = (page_directory_entry){0};
     pd[directory_index].present = attrs & 0x1;
```

```
     pd[directory_index].read_write = (attrs>>1) & 0x1;
     pd[directory_index].user_supervisor = (attrs>>2) & 0x1;
     pd[directory_index].page_table_base = ((uint32_t) pt>>12);
 }
```

Figura 15: Función auxiliar `mmu_init_page_directory_entry`

```
void mmu_init_page_table_entry(page_table_entry *pt, vaddr_t virt, paddr_t phy,
uint32_t attrs, uint32_t pages){
    uint32_t table_index = ((uint32_t) (virt>>12)) & 0x3FF; //table_indx
    for(uint32_t i = 0; i < pages; i++){
       pt[table_index + i] = (page_table_entry){0};
       pt[table_index + i].present = attrs & 0x1;
       pt[table_index + i].read_write = (attrs >> 1) & 0x1;
       pt[table_index + i].user_supervisor = (attrs >> 2) & 0x1;
       pt[table_index + i].physical_address_base = (uint32_t)(phy >> 12) + i;
    }
      }
```

Figura 16: Función auxiliar `mmu_init_page_table_entry`

```
void sched_kill_task(void){
  if(last_player == AMALIN){
      tareas_vivas_amalin[last_amalin_task] = 0;
      vivas_amalin += -1;
      kill_clock(AMALIN, last_amalin_task);
      print_ground(get_position(AMALIN, last_amalin_task));
  }else{
      tareas_vivas_betarote[last_betarote_task] = 0;
      vivas_betarote += -1;
      kill_clock(BETAROTE, last_betarote_task);
      print_ground(get_position(BETAROTE, last_betarote_task));
  }
  return;
}
```

Figura 28: Función `sched_kill_task`*, encargada de matar la tarea actual*

```
void load_map(void){
    screen[0][0].c = *"X";
    screen[0][0].a = 0x10;
    for(int j = 0; j < COL_SIZE; j++){
       for(int i = 0; i < ROW_SIZE; i++){
             char aux_char = mapa_original[j][i];
             screen[j][i].c = aux_char;
             mapa_mod[j][i] = aux_char;
             if(aux_char != *"."){
                   if(aux_char == *"P"){
                         screen[j][i].a = 0x44;
                   }else{screen[j][i].a = 0x11;}
             }else{screen[j][i].a = 0x22;}
       }
```

```
    }
}
```

Figura 30: Función que carga el mapa en pantalla

```
void print_scores(void){
      // Amalin
      /* Nombre equipo*/
      print("Amalin", 5, 41, 0x04);
      /* Caja puntajes */
      screen_draw_box(43,5,3,9,0,0x47);
      print_dec(lemSize_A, 7, 6, 44, 0x47);
      // Betarote
      /* Nombre equipo*/
      print("Betarote", 66, 41, 0x01);
      /* Caja puntajes */
      screen_draw_box(43,66,3,9,0,0x17);
      print_dec(lemSize_B, 7, 67, 44, 0x17);
}
```

Figura 31: Función que imprime los puntajes en pantalla

```
void print_teams(void){
      // Amalin
      /* Numero de leming y clock*/
      for(uint8_t i = 0; i < 5 ; i++){
      print_dec(i+1, 2, 19+i*4, 43, 0x04);
      print("x", 19+i*4, 45, 0x0F);
      }
      // Betarote
      /* Numero de leming y clock*/
      for(uint8_t i = 0; i < 5 ; i++){
      print_dec(i+1, 2, 43+i*4, 43, 0x01);
      print("x", 43+i*4, 45, 0x0F);
      }
}
```

Figura 32: Función que imprime en pantalla los equipos y los relojes de lemmings

```
move_result_e move_main(direction_e direction){
      move_result_e res = SUCCESS;
      int8_t team = sched_actual_team();
      int8_t task = sched_actual_task(team);
      pos_t *pos;
      if(team == AMALIN){
      pos = &pos_A[task];
      }else{
      pos = &pos_B[task];
      }
      res = move_valido(direction, pos);
      if(res == SUCCESS){

      print_ground(pos);
      switch (direction){
            case UP:
            pos->x=pos->x-1;
```

```
            break;
            case RIGHT:
            pos->y=pos->y+1;
            break;
            case DOWN:
            pos->x=pos->x+1;
            break;
            case LEFT:
            pos->y=pos->y-1;
            break;
      }
      print_leming(pos, team);
      }
      return res;
}
```

Figura 34: Atención a la interrupción del Syscall move

```
move_result_e move_valido(direction_e direction, pos_t *pos){
      move_result_e res = SUCCESS;
      switch(direction){
      case UP:
            res = move_up(pos);
            break;
      case RIGHT:
            res = move_right(pos);
            break;
      case DOWN:
            res = move_down(pos);
            break;
      case LEFT:
            res = move_left(pos);
            break;
      }
      return res;
}
```

Figura 35: Función encargada de verificar si el movimiento es válido según el tipo de movimiento

```
uint8_t sched_control_lemmings(void){
      pos_t pos;
      uint8_t res = 0;
      if(clock_actual%50/*401*/ == 0){
      pos.x = BASE_AMALIN_X;
      pos.y = BASE_AMALIN_Y;
      if(vivas_amalin < PLAYER_TASKS && get_simbol(&pos)  == *"."){
      uint8_t idx = 0;
      while(tareas_vivas_amalin[idx] != 0){
            idx++;
      }
      tareas_vivas_amalin[idx] = 1;
      pos_init(&pos, AMALIN, idx);
```

```
      vivas_amalin++;
      turn_amalin_created[idx] = 0;
      tss_task_reset(AMALIN, idx);
      print("|", 19+idx*4, 45, 0x0F);
      }
      pos.x = BASE_BETAROTE_X;
      pos.y = BASE_BETAROTE_Y;
      if(vivas_betarote < PLAYER_TASKS && get_simbol(&pos) == *"."){
      uint8_t idx = 0;
      while(tareas_vivas_betarote[idx] != 0){
            idx++;
      }
      tareas_vivas_betarote[idx] = 1;
      pos_init(&pos, BETAROTE, idx);
      vivas_betarote++;
      turn_betarote_created[idx] = 0;
      tss_task_reset(BETAROTE, idx);
      print("|", 43+idx*4, 45, 0x0F);
      }
      }
      clock_actual++;
      if(clock_actual%2001 == 0){
      if(vivas_amalin == PLAYER_TASKS){
      res += sched_kill_old(AMALIN);
      }
      if(vivas_betarote == PLAYER_TASKS){
      res += sched_kill_old(BETAROTE);
      }
      }
      return res;
}
```

Figura 45: Función encargada de la lógica de creación de lemmings

```
uint8_t sched_kill_old(uint8_t team){
  uint8_t res = 0;
  int8_t old_lemming = -1;
  uint32_t max_turns = 0;
  uint32_t *turns_lemming;
  if(team == AMALIN){
      turns_lemming = turn_amalin_created;
  }else{
      turns_lemming = turn_betarote_created;
  }
  for(int i = 0; i<PLAYER_TASKS; i++){
      if(max_turns < turns_lemming[i]){
      old_lemming = i;
      max_turns = turns_lemming[i];
      }
  }
  if(old_lemming != -1){
      if(old_lemming == sched_actual_task(team) && team == sched_actual_team()){
      //breakpoint();
```

```
      res = 1;
      }
      sched_explode(team, old_lemming);
      print_ground(get_position(team, old_lemming));
  }
  return res;
}
```

Figura 46: Función encargada de la lógica de creación de lemmings