DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming - Zombi Defense

Organización del Computador II
Primer Cuatrimestre de 2017

Grupo: Guiso and RT NaN JE fun

Integrante	LU	Correo electrónico
Juan Lanuza	770/15	juan.lanuza3@gmail.com
Agustin Penas	668/14	agustinpenas@gmail.com
Fernando Frassia	340/13	ferfrassia@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

Introducción

En este trabajo práctico aplicaremos los conceptos de System Programming vistos en las clases prácticas y teóricas de la materia. Un sistema operativo funciona como el nexo entre los recursos de hardware y los programas de nivel usuario. Por ejemplo, se encarga de manejar los dispositivos de entrada-salida (I/O) y la memoria del sistema.

En este trabajo construiremos un sistema operativo multitarea, con paginación y en modo protegido de 32 bits. Para correrlo contamos con el programa Bochs, que permite simular una computadora IBM-PC y realizar tareas de debugging.

El sistema es un juego con dos jugadores y cada uno de ellos lanza zombies (tareas) de su lado de la pantalla hacia el otro. Gana el jugador que logre que una mayor cantidad de tareas llegaran al lado opuesto de la pantalla.

El TP se divide en un conjunto de ejercicios:

- **Ejercicio 1:** Inicialización de la GDT y pasaje a modo protegido.
- **Ejercicio 2:** Completar las entradas de la IDT
- **Ejercicio 3:** Activar paginación
- **Ejercicio 4:** Funciones de mapeo y desmapeo de páginas y funcion de inicializar directorio y tablas de página de un zombie.
- **Ejercicio 5:** Manejo de interrupciones de teclado, reloj y de software.
- **Ejercicio 6:** Inicialización de tss de las tareas y de sus descriptores en la GDT.
- **Ejercicio 7:** Implementación del scheduler, de la syscall moverse y del modo debug.

1. Ejercicio 1

Inicialización de la GDT

Inicializamos la Tabla de Descriptores Globales con entradas para segmentos de codigo de nivel 0 y 3, otras para segmentos de datos de nivel 0 y 3 y una para un segmento que describe el area de la pantalla de video. Los segmentos de datos y códigos estan organizados de tal forma que se superpongan direccionando los primeros 623MB de memoria (Sistema FLAT), utilizando bloques de 4K al setear el bit de granularidad en 1.

A continuación explicamos cómo completamos cada uno de los campo de las entradas de la gdt:

- **Base y limite** Como mencionamos anteriormente, los segmentos de codigo y datos están superpuestos. Comienzan en la direccion base 0x00000000. El valor del límite 0x26348 corresponde la cantidad de bloques de 4k, este valor nos da los 623 MB que queremos.
- **Tipo:** El tipo para los segmentos de codigo es 0x08 (executable), mientras que para los de datos y video es 0x02 (readable, writable).
- **Sistema:** El bit de system esta seteado en 1 salvo en los segmentos correspondientes a las tss de las tareas, donde esta activo bajo en 0 pues son potestad exclusiva del sistema operativo (veremos las tss mas adelante).
- **DPL:** Los segmentos de datos y codigo de nivel kernel tienen DPL en 0x00, al igual que los segmentos de sistema y el de video, mientras que los de codigo y datos nivel usuario tienen DPL en 0x03.
- **Granularidad:** El bit de G esta activo sólo en los segmentos de datos y código ya que es necesario para tener segmentos del tamaño indicado en la consigna.
- **P, L, D/B, AVL::** Seteados en 1, 0, 1 y 0 respectivamente para todas las entradas.

Para representar a la GDT tenemos un arreglo de gdt_entry, donde gdt_entry es la siguiente estructura:

Código 1: Estructura de gdt_entry

```
typedef struct str_gdt_entry {
    unsigned short  limit_0_15;
    unsigned short  base_0_15;
    unsigned char   base_23_16;
    unsigned char   type:4;
    unsigned char   s:1;
    unsigned char   dpl:2;
    unsigned char   p:1;
    unsigned char   limit_16_19:4;
    unsigned char   avl:1;
    unsigned char   l:1;
    unsigned char   db:1;
    unsigned char   g:1;
    unsigned char   base_31_24;
} __attribute__((__packed__, aligned (8))) gdt_entry;
```

Pasaje a modo protegido

Para pasar a modo protegido completamos y cargamos la GDT con la instrucción lgdt, que toma el descriptor de la GDT dado por el siguiente struct:

Código 2: Estructura de gdt_descriptor

```
typedef struct str_gdt_descriptor {
    unsigned short  gdt_length;
    unsigned int    gdt_addr;
} __attribute__((__packed__)) gdt_descriptor;
```

Luego habilitamos A20 para tener acceso a direcciones superiores a 2^{20}, seteamos el bit PE de CR0 y saltamos a 0x40:mp. 0x40 es el selector de segmento de código de nivel 0.

Código 3: Bloque para saltar a modo protegido

```
; Habilitar A20
call habilitar_A20

; Cargar la GDT
lgdt [GDT_DESC]
; Setear el bit PE del registro CR0
MOV eax,cr0
OR eax,1
MOV cr0,eax
; Saltar a modo protegido
jmp 0x40:mp
```

Luego procedemos a setear los selectores de segmentos de datos de nivel 0 (indice 9 de la gdt seguido de 3 0 correspondientes a TI y RPL). Luego el selector de segmentos fs (indice 12). Por ultimo establecemos la pila en 0x27000.

Código 4: Bloque para setear los selectores de segmento

```
mp:
    xor eax, eax
    mov ax, 0x48 ;1001000b
    mov ds, ax
    mov es, ax
    mov gs, ax
    mov ss, ax
    mov ax, 1100000b
    mov fs, ax

; Establecer la base de la pila
    mov esp, 0x27000
```

2. Ejercicio 2

La IDT es representada por un arreglo de idt_entry (esturctura descripta en $Codigo\ 5$) de tamaño 255.

Código 5: Estructura de idt_entry

```
typedef struct str_idt_entry_fld {
    unsigned short offset_0_15;
    unsigned short segsel;
    unsigned short attr;
    unsigned short offset_16_31;
} __attribute__((__packed__, aligned (8))) idt_entry;
```

Para inicializar la IDT se hace el llamado a idt_inicializar, escrita en C. En esta función lo que hacemos es inicializar todas las entradas de la 0 a la 19, la 32, 33 y 102 utilizando la macro brindada por la cátedra. El selector de segmento está definido como el 0x40, correspondiente al codigo de kernel. Los atributos son definidos como 0x8E00 que representan segmento presente (P=1), nivel de privilegio 0 (DPL = 00), 01110 en los bits del 8 al 12 que representan una Interrupt Gate de 32 bits y el resto de los bits en 0 (del 7 al 0).

Código 6: Codigo del macro utilizado para inicializar la IDT

```
#define IDT_ENTRY_SYSTEM(numero)
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero)
        & (unsigned int) 0xFFFF);
    idt[numero].segsel = (unsigned short) 0x0040;
    idt[numero].attr = (unsigned short) 0x8e00;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero)
         >> 16 & (unsigned int) 0xFFFF);
```

Las rutinas de atencion estan dadas por una macro que llama a una función en C que recibe el numero de interrupción e imprime un mensaje dependiendo de que error es, luego queda en un loop infinito. Estas son rutinas dummy que luego fueron modificadas.

Luego de inicializar la IDT, la cargamos mediante la ejecución de la instrucción $lidt[IDT\_DESC]$. IDT_DESC es una estructura que tiene el tamaño de la tabla y la direccion de memoria de la IDT.

Código 7: Estructura de IDT_Desc

```
typedef struct str_idt_descriptor {
    unsigned short  idt_length;
    unsigned int    idt_addr;
} __attribute__((__packed__)) idt_descriptor;
```

3. Ejercicio 3

Inicializar directorio y tablas de página para el Kernel

Antes de activar paginación debemos inicializar el directorio y tablas de pagina del kernel. Esto lo haremos con identity mapping de las direcciones 0x00000000 a 0x003FFFFF. El directorio de paginas se posicionará en la dirección 0x27000.

Para representar al directorio utilizamos un struct llamado *page_directory_entry* y para las tablas utilizamos al struct *page_table_entry*. A continuación las definiciones de estos structs:

Código 8: Estructura de page_directory_entry

```
typedef struct str_page_directory_entry {
    unsigned char   p:1;
    unsigned char   rw:1;
    unsigned char   s:1;
    unsigned char   pwt:1;
    unsigned char   pcd:1;
    unsigned char   a:1;
    unsigned char   i:1;
    unsigned char   ps:1;
    unsigned char   g:1;
    unsigned char   disp:3;
    unsigned int    base:20;
} __attribute__((__packed__, aligned (4))) page_directory_entry;
```

Código 9: Estructura de page_table_entry

```
typedef struct str_page_table_entry {
    unsigned char   p:1;
    unsigned char   rw:1;
    unsigned char   s:1;
    unsigned char   pwt:1;
    unsigned char   pcd:1;
    unsigned char   a:1;
    unsigned char   d:1;
    unsigned char   pat:1;
    unsigned char   g:1;
    unsigned char   disp:3;
    unsigned int    base:20;
} __attribute__((__packed__, aligned (4))) page_table_entry;
```

Para inicializar ambas creamos un puntero a $page\_directory\_entry$ y lo seteamos en 0x27000; luego un puntero a $page\_table\_entry$ y lo inicializamos en 0x28000.

Código 10: Inicialización de los punteros

```
page_directory_entry* dir_pagina_kernel = (page_directory_entry*) CR3KERNEL;
page_table_entry* dir_table_kernel = (page_table_entry*) 0x28000;
```

Luego inicializamos el directorio con los siguientes parametros:

Código 11: Inicialización del directorio

```
dir_pagina_kernel->p = 1;
dir_pagina_kernel->rw = 1;
dir_pagina_kernel->s = 0;
dir_pagina_kernel->pwt = 0;
dir_pagina_kernel->pcd = 0;
dir_pagina_kernel->a = 0;
dir_pagina_kernel->i = 0;
dir_pagina_kernel->ps = 0;
dir_pagina_kernel->g = 0;
dir_pagina_kernel->disp = 0;
dir_pagina_kernel->base = 0x28;
```

Como el Kernel ocupará el primer MB, tendremos 1024 tablas (1024 x 4kb) y cada una de ellas la inicializaremos de la siguiente manera:

Código 12: Inicialización de las tablas

```
dir_table_kernel[i].p = 1;
dir_table_kernel[i].rw = 1;
dir_table_kernel[i].s = 0;
dir_table_kernel[i].pwt = 0;
dir_table_kernel[i].pcd = 0;
dir_table_kernel[i].a = 0;
dir_table_kernel[i].d = 0;
dir_table_kernel[i].pat = 0;
dir_table_kernel[i].g = 0;
dir_table_kernel[i].disp = 0;
dir_table_kernel[i].base = i;
```

Activar paginación

Una vez que tenemos inicializado el mapeo del Kernel podemos pasar a activar paginación. Para esto debemos activar el bit de PG del registro CR0:

Código 13: Inicialización de las tablas

```
mov eax , cr0
or eax , 0 x80000000
mov cr0 , eax
```

Limpiar buffer de video

La limpieza del buffer de video y el dibujo del mapa fue hecho utilizando las funciones de print brindadas por la catedra. Estas funciones se encargan de escribir en la memoria del video el string, o número, pasado por parametro.

4. Ejercicio 4

Inicialización de la MMU

Al inicializar la MMU, tenemos dos tareas que hacer: inicializar la dirección donde se encuentra la próxima página libre y mapear el directorio del kernel(descripto en el Ejercicio 3). Luego, a medida que se pide la siguiente página, aumentamos esta variable en 4kb (una página) para obtener la dirección de la siguiente.

Código 14: Contador de Páginas Libres

```
unsigned int proxima_pagina_libre;

void mmu_inicializar() {
  inicializar_kernel_mapping();
  proxima_pagina_libre = INICIO_PAGINAS_LIBRES;
}

unsigned int mmu_proxima_pagina_fisica_libre() {
    unsigned int pagina_libre = proxima_pagina_libre;
    proxima_pagina_libre += PAGE_SIZE;
    return pagina_libre;
}
```

Mapeo y Desmapeo de Páginas

Los algoritmos de mapeo y desmapeo se pueden entender con mayor facilidad con el siguiente diagrama:

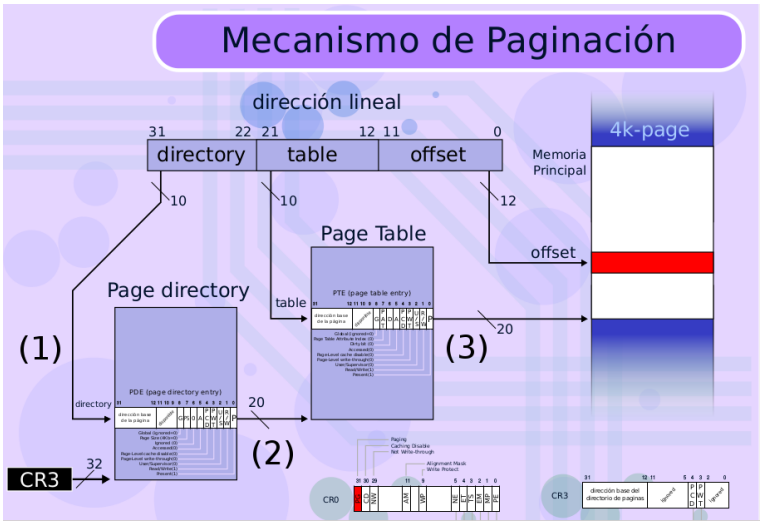

Figura 1: Mapeo de Páginas

Para mapear páginas `mmu_map_page` toma una dirección virtual, una física, un selector del directorio a mapear, y dos parámetros de permisos (user/supervisor, read/write).

El algoritmo de mapeo es el siguiente:

- Primero limpiamos los bits de permisos del selector y obtenemos el directorio.
- Luego buscamos la tabla de páginas para mapear. Para esto avanzamos hasta su descriptor sumando a la dirección base el offset indicado en los primeros 10 bits de la dirección virtual.
- Si dicho descriptor tiene el bit presente en 1, shifteamos su base y obtenemos la dirección del comienzo de la tabla que buscamos; sino, creamos la tabla y asignamos su base al descriptor del directorio.

- Finalmente avanzamos hasta el descriptor de la página, seteamos el bit de presente junto con los demás permisos y su base será la dirección física shifteada 12 a la derecha.

Código 15: Mapeo de páginas

```
void mmu_map_page(unsigned int virtual, unsigned int cr3, unsigned int fisica,
    char sup, char rw){
    page_directory_entry* page_dir = (page_directory_entry*) ((cr3 >> 3) << 3);

    page_table_entry* pte;
    if (page_dir[PDE_INDEX(virtual)].p){
        pte = (page_table_entry*) (page_dir[PDE_INDEX(virtual)].base << 12);
    } else {
        pte = (page_table_entry*) mmu_proxima_pagina_fisica_libre();
        page_dir[PDE_INDEX(virtual)].base = ((unsigned int) pte) >> 12;
        page_dir[PDE_INDEX(virtual)].p = 1;
        page_dir[PDE_INDEX(virtual)].rw = 1;
        page_dir[PDE_INDEX(virtual)].s = 1;
    }
    pte[PTE_INDEX(virtual)].p = 1;
    pte[PTE_INDEX(virtual)].rw = rw;
    pte[PTE_INDEX(virtual)].s = sup;
    pte[PTE_INDEX(virtual)].pwt = 0;
    pte[PTE_INDEX(virtual)].pcd = 0;
    pte[PTE_INDEX(virtual)].a = 0;
    pte[PTE_INDEX(virtual)].d = 0;
    pte[PTE_INDEX(virtual)].pat = 0;
    pte[PTE_INDEX(virtual)].g = 0;
    pte[PTE_INDEX(virtual)].disp = 0;
    pte[PTE_INDEX(virtual)].base = fisica >> 12;
    tlbflush();
}
```

`mmu_unmap_page` toma la dirección virtual y el cr3 del directorio. Para desmapear accedemos al descriptor de dicha página y seteamos el bit de presente en 0.

Código 16: Desmapeo de páginas

```
void mmu_unmap_page(unsigned int virtual, unsigned int cr3){
    page_directory_entry* page_dir = (page_directory_entry*) ((cr3 >> 3) << 3);
    page_table_entry* pte = (page_table_entry*) (page_dir[PDE_INDEX(virtual)].
        base << 12);
    pte[PTE_INDEX(virtual)].p = 0;
}
```

Inicialización de directorios y tablas para Zombis

`mmu_inicializar_dir_zombi` toma el tipo de zombie que el jugador quiere inicializar y la posición (x,y) donde se quiere inicializarlo y devuelve la dirección del directorio creado para este zombi.

La inicialización del directorio y tabla de páginas para dicho zombi respetan el formato presentado en el enunciado:

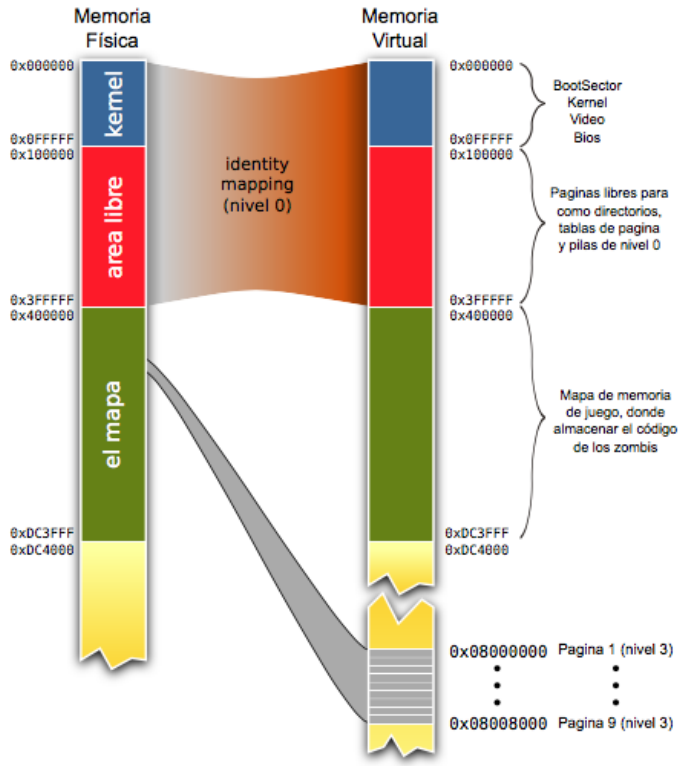

Figura 2: Mapa de memoria de la Tarea.

La descripción del algoritmo es la siguiente:

- Para obtener el nuevo directorio, pedimos una página libre y mapeamos con identity mapping todas las páginas entre 0x000000 y 0x03FFFFF.
- Luego, obtenemos la dirección física a la que corresponde esa posición (entre 0x400000 y 0xDC3FFF) y copiamos el código del zombi (que dependiendo del tipo, está en alguna dirección física fija). Cabe aclarar que primero mapeamos en el directorio del kernel la página destino (que no está mapeada), y luego de copiar el código la desmapeamos.
- Por último, mapeamos las 9 páginas virtuales del zombi a sus correspondientes físicas.

Código 17: Inicialización directorio y tabla de páginas del zombi

```
unsigned int mmu_inicializar_dir_zombi(unsigned short x, unsigned short y,
    zombie z) {
    page_directory_entry* dir_pagina = (page_directory_entry*)
        mmu_proxima_pagina_fisica_libre();
    inicializar_con_identity_mapping(dir_pagina);

    unsigned int* origin;
    switch (z) {
        case A_MONK:
            origin = (unsigned int*)0x10000;
            break;
        case A_SUICIDE_UNIT:
            origin = (unsigned int*)0x11000;
            break;
        case A_DRUNK_DRIVER:
            origin = (unsigned int*)0x12000;
            break;
        case B_MONK:
            origin = (unsigned int*)0x13000;
            break;
        case B_SUICIDE_UNIT:
            origin = (unsigned int*)0x14000;
            break;
        case B_DRUNK_DRIVER:
            origin = (unsigned int*)0x15000;
            break;
    }

    unsigned int* dest = (unsigned int*) mmu_get_map_position(x, y);
    mmu_map_page((unsigned int)dest, rcr3(), (unsigned int)dest, 0, 1);
    unsigned int i;
    for (i=0; i<1024; i++){
        dest[i] = origin[i];
    }

    mmu_unmap_page((unsigned int)dest, rcr3());

    if (x == POS_INIT_ZOMBI_A){
        mmu_map_adjacent_to_zombi(0, (unsigned int) dir_pagina, x, y);
    } else if (x == POS_INIT_ZOMBI_B){
        mmu_map_adjacent_to_zombi(1, (unsigned int) dir_pagina, x, y);
    }

    return (unsigned int)dir_pagina;
}
```

Para manejar el video convenientemente, nos pareció útil hacer una función para obtener la página del video asociada a una coordenada.

Código 18: Obtener página de coordenada

```
unsigned int mmu_get_map_position ( int x,  int y) {
    return 0x400000 + (MOD(y,44) * 78 + x) * PAGE_SIZE;
}
```

Finalmente, `mmu_map_adjacent_to_zombi` mapea las 9 páginas virtuales del zombie a las 9 páginas físicas del video.

Código 19: Mapeo de zombi a una coordenada en el mapa

```
void mmu_map_adjacent_to_zombi(unsigned int player, unsigned int dir_pagina,
    unsigned int x, unsigned int y){
    mmu_map_page(VIRTUAL_COD_ZOMBIE_1, dir_pagina, mmu_get_map_position(x, y),
        1, 1);
    if (player == 0) {
        mmu_map_page(VIRTUAL_COD_ZOMBIE_2, dir_pagina, mmu_get_map_position(x+1,
            y), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_3, dir_pagina, mmu_get_map_position(x+1,
            y+1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_4, dir_pagina, mmu_get_map_position(x+1,
            y-1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_5, dir_pagina, mmu_get_map_position(x, y
            +1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_6, dir_pagina, mmu_get_map_position(x, y
            -1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_7, dir_pagina, mmu_get_map_position(x-1,
            y), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_8, dir_pagina, mmu_get_map_position(x-1,
            y-1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_9, dir_pagina, mmu_get_map_position(x-1,
            y+1), 1, 1);
    } else {
        mmu_map_page(VIRTUAL_COD_ZOMBIE_2, dir_pagina, mmu_get_map_position(x-1,
            y), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_3, dir_pagina, mmu_get_map_position(x-1,
            y-1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_4, dir_pagina, mmu_get_map_position(x-1,
            y+1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_5, dir_pagina, mmu_get_map_position(x, y
            -1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_6, dir_pagina, mmu_get_map_position(x, y
            +1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_7, dir_pagina, mmu_get_map_position(x+1,
            y), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_8, dir_pagina, mmu_get_map_position(x+1,
            y+1), 1, 1);
        mmu_map_page(VIRTUAL_COD_ZOMBIE_9, dir_pagina, mmu_get_map_position(x+1,
            y-1), 1, 1);
    }
}
```

5. Ejercicio 5

Asociamos las rutinas de interrupcion a la de reloj, teclado y syscall

Para inicializar las interrupciones de teclado y reloj utilizamos la misma macro nombrada en el Ejercicio2, llamada IDT_ENTRY_SYSTEM. Para inicializar la entrada de la syscall creamos una macro parecida a la IDT_ENTRY_SYSTEM pero con distintos atributos. A esta la llamaremos IDT_ENTRY_USER y la describimos de la siguiente manera:

Código 20: Macro IDT_ENTRY_USER

```
#define IDT_ENTRY_USER(numero)
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero)
        & (unsigned int) 0xFFFF);
    idt[numero].segsel = (unsigned short) 0x0040;
    idt[numero].attr = (unsigned short) 0xEE00;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero)
         >> 16 & (unsigned int) 0xFFFF);
```

Rutina atención de clock

Creamos la rutina de atención de clock, en esta llamamos a próximo reloj que se encargá de avanzar el dibujo del clock y luego avisamos al PIC que la interrupción fue atendida mediante call fin_intr_pic1. Luego será modificada para el uso del scheduler.

Código 21: Rutina atención clock

```
_isr32:
    pushad
    call proximo_reloj
    call fin_intr_pic1
    popad
    iret
```

Rutina de atención de teclado

Creamos la rutina de atención de teclado. Esta rutina toma el valor de la tecla presionada y compara este valor con los de las teclas correspondientes al juego. Luego hace un jump a el código de esa tecla en particular. A continuación un ejemplo de la tecla a:

Código 22: Rutina atención teclado

```
_isr33:
    pushad
    xor eax, eax
    in al, 0x60

    cmp al, 0x1e
    je .a

.a: ; A hacia izq
    mov ebx, -1
    push ebx
    mov ebx, 0
    push ebx
    call game_change_zombie
    add esp, 8
    jmp .fin
```

```
    .fin:
        call fin_intr_pic1
        popad
        iret
```

El código real es más largo, contiene comparaciones y etiquetas para todas las teclas que son parte del juego.

Rutina atención syscall

La siguiente es la rutina de atención de la interrupción 0x66, este es el código ya modificado para atender la syscall. Luego de mover al zombie hacemos un salto a la tarea idle.

Código 23: Rutina atención de syscall

```
_isr102:
    pushad
    push eax
    call game_move_current_zombi
    pop eax
    ;jump a la tarea idle
    jmp 0x70:0xFAFAFA
    popad
    iret
```

6. Ejercicio 6

Inicialización de descriptores de TSS en la GDT

Para inicializar los Task State Segment (TSS) debemos agregar sus descriptores a la GDT. La mayoría de las variables del descriptor de TSS (gdt_entry) son iguales para todas las TSS:

Código 24: Descriptor de TSS (gdt_entry)

```
gdt[GDT_IDX_TAREA] = (gdt_entry) {
        (unsigned short)    0x0068,         /* limit[0:15]  */
        (unsigned short)    0x0000,         /* base[0:15]   */
        (unsigned char)     0x00,           /* base[16:23]  */
        (unsigned char)     0x09,           /* type         */
        (unsigned char)     0x00,           /* s            */
        (unsigned char)     0x00,           /* dpl          */
        (unsigned char)     0x00,           /* p            */
        (unsigned char)     0x00,           /* limit[16:19] */
        (unsigned char)     0x00,           /* avl          */
        (unsigned char)     0x00,           /* l            */
        (unsigned char)     0x00,           /* db           */
        (unsigned char)     0x00,           /* g            */
        (unsigned char)     0x00,           /* base[24:31]  */
    }
```

A diferencia de las tareas-zombie, la Idle y la inicial comienzan con el bit de presente en 1. Por otro lado, la base de la tss se setea en el momento de inicializar cada tarea, cuando conocemos su posicion en memoria.

Inicialización de TSSs

Para representar a las tss, contamos con el siguiente struct:

Código 25: Descriptor de TSS (gdt_entry)

```
typedef struct str_tss {
    unsigned short  ptl;            unsigned short  unused0;
    unsigned int    esp0;
    unsigned short  ss0;            unsigned short  unused1;
    unsigned int    esp1;
    unsigned short  ss1;            unsigned short  unused2;
    unsigned int    esp2;
    unsigned short  ss2;            unsigned short  unused3;
    unsigned int    cr3;
    unsigned int    eip;
    unsigned int    eflags;
    unsigned int    eax;
    unsigned int    ecx;
    unsigned int    edx;
    unsigned int    ebx;
    unsigned int    esp;
    unsigned int    ebp;
    unsigned int    esi;
    unsigned int    edi;
    unsigned short  es;             unsigned short  unused4;
    unsigned short  cs;             unsigned short  unused5;
    unsigned short  ss;             unsigned short  unused6;
    unsigned short  ds;             unsigned short  unused7;
    unsigned short  fs;             unsigned short  unused8;
    unsigned short  gs;             unsigned short  unused9;
```

```
    unsigned short  ldt;            unsigned short  unused10;
    unsigned short  dtrap;          unsigned short  iomap;
} __attribute__((__packed__, aligned (8))) tss;
```

En el caso de la tss de la tarea inicial no es necesario setear ninguna variable, solo se usa para guardar el contexto al saltar a la Idle y nunca se vuelve. Para la tss de la Idle las variables se setean de acuerdo a la consigna del tp.

Por otro lado, contamos con la función tss_inicializar_zombie que se encarga inicializar una tarea lanzada por un jugador. Primeramente, busca un slot libre en el arreglo gdt_indexes_tasks en la estructura info_jugador (explicada de el ejercicio 7). Con índice del slot libre accedemos a su descriptor de segmento en la gdt para setear el bit de presente en 1 y su base a que apunte a donde esta la tss. Posteriormente inicializamos el directorio de página de la tarea y pedimos una página libre para usarla como stack de nivel 0. Por último seteamos las variables de la tss:

Código 26: Descriptor de TSS (gdt_entry)

```
unsigned int dir_z = mmu_inicializar_dir_zombi(x, y, z);
unsigned int ss = mmu_proxima_pagina_fisica_libre();
tss_zombi->esp0 = ss + PAGE_SIZE;
tss_zombi->ss0 = GDT_IDX_DAT_KERNEL<<3;
tss_zombi->cr3 = dir_z;
tss_zombi->eip = VIRTUAL_COD_ZOMBIE_1;
tss_zombi->eflags = 0x202;
tss_zombi->cs = GDT_IDX_COD_USER << 3 | 0x3;
tss_zombi->es = GDT_IDX_DAT_USER << 3 | 0x3;
tss_zombi->ss = GDT_IDX_DAT_USER << 3 | 0x3;
tss_zombi->ds = GDT_IDX_DAT_USER << 3 | 0x3;
tss_zombi->fs = GDT_IDX_DAT_USER << 3 | 0x3;
tss_zombi->gs = GDT_IDX_DAT_USER << 3 | 0x3;
tss_zombi->esp = VIRTUAL_COD_ZOMBIE_1 + PAGE_SIZE;
tss_zombi->ebp = VIRTUAL_COD_ZOMBIE_1 + PAGE_SIZE;
```

Carga de tarea inicial y salto a la Idle

Para cargar la tarea inicial tenemos que setear el task register:

```
mov ax, 0x0D<<3
ltr ax
```

Posteriormente, para pasar a la tarea Idle simplemente hacemos:

```
jmp 0x70:0xFAFAFA
```

Donde 0x70 es el selector de segmento de la tarea.

7. Ejercicio 7

Estructuras del Scheduler

Las variables de nuestro scheduler son las siguientes:

Código 27: Variables del Scheduler

```
info_player playerA;
info_player playerB;
int playerActual;
int isIdle;
```

$playerActual$ indica cuál es el jugador que tiene el turno actualmente; $info\_playerA$ e $info\_playerB$ guardan la información relevante al jugador. Finalmente, nos pareció relevante agregar la variable $isIdle$, que indica si la tarea actual es la Idle.

Código 28: Información del zombi

```
typedef struct str_info_zombie {
    zombie  type;
    unsigned short  x;
    unsigned short  y;
    unsigned short reloj_actual;
} __attribute__((__packed__, aligned (16))) info_zombie;
```

`info_zombie` guarda su posición actual la pantalla, su tipo (que es un enum), y su reloj actual (que es un número entre 0 y 3 que indica qué caracter del reloj tiene dibujado el zombi en este momento).

Código 29: Información del jugador

```
typedef struct str_info_player {
    zombie  selected_type;
    unsigned short  y;
    unsigned short cant_lanzados;
    unsigned short gdt_indexes_tasks[CANT_ZOMBIS];
    info_zombie info_zombies[CANT_ZOMBIS];
    unsigned short curr_zombie;
    unsigned short puntos;
} __attribute__((__packed__, aligned (16))) info_player;
```

Por otro lado, `info_player` guarda el tipo de zombie seleccionado actualmente, la posición del jugador en el eje vertical, la cantidad de zombies lanzados, y sus puntos. Lo más interesante es `gdt_indexes_tasks` que es un arreglo de selectores de la gdt, donde el i-ésimo selector se corresponde con el i-ésimo zombie de info_zombies. Por último, curr_zombie indica el índice del zombie actual del jugador.

Task switch

Como vemos en la figura 3, las tareas de los zombies deben ir alternandose entre el jugador A y el B. Además los zombies de cada jugador deben correr uno tras otro y de manera cíclica.

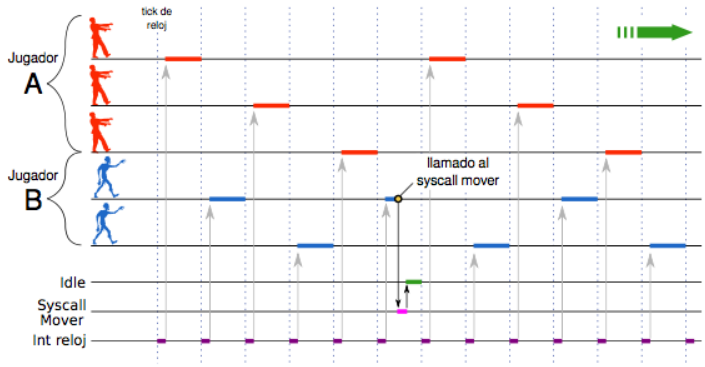

Figura 3: Ejemplo de funcionamiento del Scheduler

Lo que hacemos entonces es preguntar en cada interrupción de reloj quién es el próximo índice de tarea a ejecutar y hacemos el jump a ella. En caso que el scheduler nos devuelva el índice 0 no haremos nada, nos quedamos en la tarea actual. A continuación el código de la interrupción de clock.

Código 30: Interrupción de clock

```
_isr32:
    pushad

    call proximo_reloj
    mov eax, [inDebugMode]
    mov esi, [debugScreenOpen]
    and eax, esi
    cmp eax, 1
    je .nojump  ;si esta en modo debug con la pantalla de debug abierta no hago
        nada


    call sched_proximo_indice

    cmp ax, 0
    je .nojump
        push eax
        call girar_reloj_actual
        pop eax
        mov [selector], ax
        call fin_intr_pic1
        jmp far [offset]
        jmp .end

    .nojump:
    call fin_intr_pic1

    .end:
    popad
    iret
```

Nos salteamos las lineas que hablan del debugger ya que este será explicado más adelante. Como vemos hacemos un *call* a *sched_proximo_indice* como habiamos dicho y a continuación saltamos a esa tarea. En caso de ser 0 no hacemos ningún *jmp*.

Lo que hace $sched\_proximo\_indice$ es tomar la información de playerActual para saber quien es el siguiente jugador. Si el jugador opuesto al actual no tiene tareas en su lista de gdt_index_tasks (osea tiene todos 0), entonces no se cambiará el $playerActual$. En caso contrario, el player actual pasará a el otro jugador.

Si este es el primer zombie que se lanza (caso en que zombie inicial del player será 32) entonces verifico que haya zombies en la lista de indices, puesto que si no los hay es por que aún no se han lanzado zombies y por ende no hay que saltar a ninguna tarea de este jugador. Si hay zombies en la lista entonces el primer zombie será el proximo a saltar.

Si no estamos en el caso anterior entonces lo que hay que hacer es recorrer la lista de índices busucando el primero que sea distinto de 0. Si se llega al final del arreglo se empieza desde el inicio. Si se encuentra de esta manera una tarea distinta a la actual para este jugador entonces será ese índice el que se devuelva y esa tarea a la que se salte.

Si se dio toda la vuelta y se llegó al mismo zombie que el actual entonces tenemos que preguntarnos si estamos en la tarea Idle ya que si estamos en la tarea idle entonces vuelve a ser el turno de este zombie, pero si no estamos en la tarea idle entonces no debemos hacer nada y devolver 0 ya que actualmente estamos en el turno de ese zombie.

Syscall mover

La syscall mover es la única proveída por el sistema, y está mapeada a la interrupción 0x66.
La codificación de desplazamientos es la siguiente:

```
Adelante   =  0x83D
Atras      =  0x732
Derecha    =  0x441
Izquierda  =  0xAAA
```

Figura 4: Códigos de mover

Atendemos la interrupción en assembler y llamamos a `game_move_current_zombi` en C, cuyo único parámetro es la dirección en la que quiere moverse el zombie, y su funcionamiento es:

- 1. Dada la dirección pasada por parámetro, obtenemos la página virtual a donde se va a mover.

Código 31: Syscall mover - obtener origen y destino

```
    unsigned int* orig = (unsigned int*) VIRTUAL_COD_ZOMBIE_1;
    unsigned int* dest;
    switch (dir) {
    case IZQ:
        dest = (unsigned int*)VIRTUAL_COD_ZOMBIE_6;
        break;
    case DER:
        dest = (unsigned int*)VIRTUAL_COD_ZOMBIE_5;
        break;
    case ADE:
        dest = (unsigned int*)VIRTUAL_COD_ZOMBIE_2;
        break;
    case ATR:
        dest = (unsigned int*)VIRTUAL_COD_ZOMBIE_7;
        break;
    }
```

- 2. Copiamos la página entera del código del zombie, a su página física de destino. Esto es copiando su página virtual actual a su página virtual destino, porque ambas están mapeadas sus correspondientes físicas.

Código 32: Syscall mover - copiar código del zombie

```
    int i;
    for(i=0; i<1024; i++){
        dest[i] = orig[i];
    }
```

- 3. Calculamos la nueva posición (x, y) (tomando en cuenta qué jugador es el actual, y la circularidad del mapa).

Código 33: Syscall mover - calcular nueva (x, y)

```
info_player* current_player = get_current_player();
    unsigned short current_zombie = current_player->curr_zombie;
    unsigned short x_orig = (current_player->info_zombies[current_zombie]).x;
    unsigned short y_orig = (current_player->info_zombies[current_zombie]).y;
    unsigned short x_dst = x_orig;
```

```
unsigned short y_dst = y_orig;

if(y_dst==0 || y_dst==43){
    if ((dir == IZQ && playerActual == 0) || (dir == DER && playerActual ==
        1)) {
        y_dst=43;
    } else if ((dir == DER && playerActual == 0) || (dir == IZQ &&
        playerActual == 1)) {
        y_dst=0;
    } else if ((dir == ADE && playerActual == 0) || (dir == ATR &&
        playerActual == 1)) {
        x_dst++;

    } else if ((dir == ATR && playerActual == 0) || (dir == ADE &&
        playerActual == 1)) {
        x_dst--;
    }
}else{
    if ((dir == IZQ && playerActual == 0) || (dir == DER && playerActual ==
        1)) {
        y_dst--;
    } else if ((dir == DER && playerActual == 0) || (dir == IZQ &&
        playerActual == 1)) {
        y_dst++;
    } else if ((dir == ADE && playerActual == 0) || (dir == ATR &&
        playerActual == 1)) {
        x_dst++;

    } else if ((dir == ATR && playerActual == 0) || (dir == ADE &&
        playerActual == 1)) {
        x_dst--;
    }
}
```

- 4. Asignamos al zombie a su nueva posición.

Código 34: Syscall mover - asignar nuevas coordenadas al zombie

```
(current_player->info_zombies[current_zombie]).x = x_dst;
(current_player->info_zombies[current_zombie]).y = y_dst;
```

- 5. De llegar a un borde, sumamos punto, matamos al zombie, y terminamos la ejecución del algoritmo.

Código 35: Syscall mover - sumar punto y matar zombie

```
if(x_dst==1){
    game_sumar_punto(1/*jugador azul*/);
    game_matar_zombie_actual();
    return;
}
if(x_dst==78){
    game_sumar_punto(0/*jugador rojo*/);
    game_matar_zombie_actual();
    return;
}
```

- 6. Remapeamos todas las páginas virtuales del zombie teniendo en cuenta su nueva posición (x, y).

Código 36: Syscall mover - remapear las páginas del zombie (vease Ej. 4)

```
/* remapear sus direcciones */
mmu_map_adjacent_to_zombi(playerActual, rcr3(), x_dst, y_dst);
```

- 7. Por último imprimimos el cambio en pantalla y seteamos la tarea idle en el scheduler.

Código 37: Syscall mover - Imprimir en pantalla y setear Idle

```
print_move_zombie(playerActual, x_orig, y_orig, x_dst, y_dst, (
    current_player->info_zombies[current_zombie]).type);
isIdle=1;
```

Modo debug

Para implementar el modo debug en nuestro juego nos encontramos con tres problemáticas:

1. Cómo recuperar los valores de algunos registros que son modificados cuando se produce la interrupción
2. Guardar el estado de la pantalla del juego antes de que aparezca el cartel, para luego poder restaurarla
3. Que otras tareas no se sigan ejecutando y que los jugadores no puedan realizar acciones cuando esta la pantalla de debug abierta

Para guardar el estado de los registros al momento de producirse una interrupción contamos con el struct debug_info. La mayoría de los registros siguen iguales después de producida la interrupción. Algunos son modificados, pero antes son apilados en el stack. El estado del stack despues de transferir el control al handler de interrupciones es el siguiente:

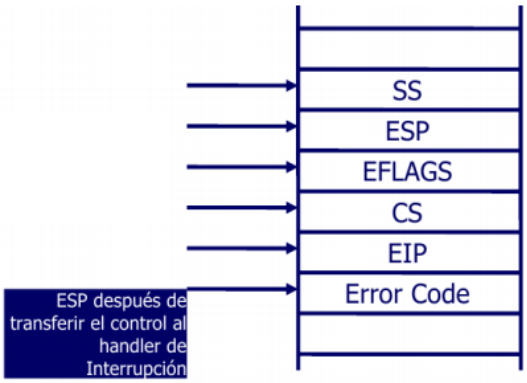

Figura 5: Stack al momento de entrar al handler de interrupciones

Por otro lado, para poder guardar el estado de la pantalla antes de que aparezca el cartel del debugger contamos con la variable $backup$ (matriz de ca que se corresponde con la pantalla), la función $backup\_screen$ (que guarda la pantalla en la variable) y la función $backup\_restore\_screen$ (que escribe en la pantalla lo que hay guardado en la variable).

Por último, para poder manejar las tareas en modo debug contamos con dos variables globales: inDebugMode, que indican si se está en modo debug, y $debugScreenOpen$, que indica si la ventana de debug está abierta. Al producirse la excepción se setea $debugSreenOpen$ en 1, se muestra la pantalla de debug y se salta a la tarea Idle. Dentro de la interrupción de clock, si $inDebugMode$ y $debugScreenOpen$ están seteados, no se salta a la próxima tarea. Dentro de la interrupción de teclado, si $inDebugMode$ y $debugScreenOpen$ están seteados, solamente la tecla 'y' produce un cambio en el juego (cerrar la pantalla de debug).

8. Conclusiones

Para concluir el informe, nos parece que este trabajo nos ayudó a entender muchos mecanismos claves de un sistema operativo, como por ejemplo el manejo de memoria, manejo de tareas y atención de interrupciones.

Y, de más está decir que sabemos que un sistema operativo real comprende una complejidad bastante superior al nuestro, pero confiamos que este trabajo nos sentó una buena base para ayudarnos a entender un sistema real.