

Resumen

En el presente trabajo se describe la problemática de ...

Índice

1. Objetivos generales	2
2. Resolución de los Ejercicios	2
2.1. Ejercicio 1	2
2.1.1. Consideraciones de ej 1	3
2.2. Ejercicio 2	4
2.3. Ejercicio 3	4
2.4. Ejercicio 4	6
2.5. Ejercicio 5	9
2.6. Ejercicio 6	9
2.7. Ejercicio 7	13
3. Aclaraciones del Makefile/Generales	15



Figura 1: Descripción de la figura

1. Objetivos generales

El objetivo de este Trabajo Práctico es el de conocer y desarrollar un mejor entendimiento de la organización interna del computador y su evolución, así como el manejo de la estructura que esta organización nos provee, mediante el modelamiento práctico de un juego desde el modo real de la máquina.

2. Resolución de los Ejercicios

2.1. Ejercicio 1

En el ejercicio 1 se completa e inicializa la GDT, con cuatro segmentos, 2 de código (uno del kernel y el otro del usuario), y dos de datos (uno del kernel y otro del usuario), dirigiendo los primeros 878MB de memoria. (Se dejan las primeras 3 posiciones de la gdt libre, una o sea la nula y las otras dos por restricciones del tp). El primer índice que deben usar para declarar los segmentos es el 4 (contando desde 0).

Para completar la gdt, agregamos los descriptores de segmento a la GDT, modificando el archivo gdt.c. Allí, describimos los segmentos completando estructuras de descriptores y descriptor de gdt (str_gdt_entry, y str_gdt_descriptor, definidas en gdt.h),

Formato de str_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;
```

Allí, al nulo se le pone todo 0 y a los otros se le pone diferentes parámetros dependiendo si es un

descriptor de código, de datos, o si es un descriptor de sistema. Por ejemplo, el descriptor de código del kernel que definimos nos queda:

```
[GDT_IDX_KERNEL_CODE_DESC] = (gdt_entry) {
    (unsigned short) SEG_LIMIT_0_15(0x36E00), /* limit[0:15] */
    (unsigned short) SEG_BASE_0_15(0x0),      /* base[0:15] */
    (unsigned char)  SEG_BASE_16_23(0x0),      /* base[23:16] */
    (unsigned char)  SEG_CODE_EXRD,           /* type */
    (unsigned char)  SEG_TYPE_CODEDATA,       /* s */
    (unsigned char)  SEG_PRIV0,               /* dpl */
    (unsigned char)  SEG_PRESENT,             /* p */
    (unsigned char)  SEG_LIMIT_16_19(0x36E00), /* limit[16:19] */
    (unsigned char)  SEG_AVL,                 /* avl */
    (unsigned char)  !SEG_IA32E,              /* l */
    (unsigned char)  SEG_MOD_32b,             /* db */
    (unsigned char)  SEG_GRAN_4K,             /* g */
    (unsigned char)  SEG_BASE_24_31(0x0),      /* base[31:24] */
}
```

donde todos los inputs son defines que se encuentran en el archivo defines.h

2.1.1. Consideraciones de ej 1

- Se flatea las secciones, poniéndoles a todas la misma dirección base (0x00) y de límite se coloca el tamaño-1 / 0x400 + 0x3FF (vamos a poner el de granularidad en 1, para que la cuenta nos de el tamaño pedido menos 1)
- El tipo es read/write(0x02) en los de datos, y (execute/read) en los de código.
- El s es 1 x ser de datos o códigos.
- la dpl es 3 o 0 dependiendo si es un segmento de usuario o del kernel.
- el l está en 0 x estar en 32 bytes. el db está en 1 por el mismo motivo.

También hay un segmento de video (del kernel) cuya dirección de entrada es la pedida por la cátedra (y límite de acuerdo al tamaño de la pantalla pedida.), el resto es como un segmento de código definido antes.

En el kernel.asm, para pasar a modo protegido se pone la directiva BITS 16, (para que el linker sepa que se interpreta en 16 bits las direcciones) Luego, se desactivan las interrupciones (cli), se cambia el modo de video (interrupción) (se va a modo 3h y luego se setea)

Se habilita la A20 (con una función) y se carga la gdt con la función lgdt (y la dirección de la tabla (y tamaño)).

Luego se pasa a modo protegido seteando el bit PE del registro CR0. y se salta a la siguiente instrucción, (desde seg de código de la gdt), que es la siguiente (en el medio, se usa la directiva BITS 32, para que reconozca que trabajamos con 32 bits).

Se establecen los selectores de segmento (ds, es, fs, gs, ss) (todos apuntan al segmento de datos del kernel, porque es el código del kernel) Con estos segmentos seteados, podemos establecer la pila moviendo la base a los reg ebp y esp (dirección pedida por cátedra)

Luego se inicializa la pantalla, para ello llamamos a la función inicializar_pantalla, que pushea ds, mueve eax al segmento de video, (y a la primera posición), y mientras el contador ecx lopee (tiene la cantidad de words a pintar de gris), avanzamos eax y por cada 2 bytes llenamos el lugar del color querido (01110000b ; 0111 = grey sin bright, 0000 = black sin bright). Luego popeamos ds y salimos.

Terminamos ejercicio 1

2.2. Ejercicio 2

Tenemos que completar las entradas de la IDT para asociar las diferentes rutinas a todas las llamadas del procesador. La IDT (Interrupt Descriptor Table), es la que almacena los descriptores de interrupciones (descriptores de sistema que pueden ser de tres tipos, trap, interrupt o task, nosotros vamos a usar interrupt solamente). Para completar la IDT llamamos desde el kernel.asm a `idt_inicializar`. Esta función contiene los descriptores que serán puestos en la GDT (lo que hacemos utilizando la instrucción `lidt [IDT_DESC]`, donde `IDT_DESC` es el límite y la base de la IDT.)

Los descriptores de interrupción están armados de la siguiente manera:

```
#define IDT_ENTRY(numero)    /*es un descriptor de interrupcion de nivel kernel*/
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) );
    idt[numero].segsel = (unsigned short) 0x18;
    idt[numero].attr = (unsigned short) IDT_PRESENT | IDT_USR0 | IDT_INTERRUPT | IDT_32BITS;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16);
```

En las entradas de la IDT (`IDT_ENTRY`), colocamos el offset de la dirección de la rutina de atención de interrupciones (definidas como `isr(numero de interrupción)`).

Las rutinas de interrupción consisten en imprimir infinitamente el texto el nombre de la excepción que el procesador generó, en la pantalla (en la parte superior derecha); En bits correspondientes de los atributos del descriptor de interrupciones, seteamos:

- que este presente.
- que tenga prioridad 0 porque son excepciones del procesador
- aclaramos el tipo (interrupt)
- que sea de 32bits

El `segsel` es el 0x18 por ser el tercero de la gdt, donde está el descriptor del segmento donde se encuentran las rutinas de atención de interrupciones.

Cuando se produce una interrupción, el procesador busca en la `istr` la `idt`, va a la `Idt` y se va a la puerta de esa interrupción. Se usa el segmento para ir a la entrada de la gdt en donde está el descriptor, se le suma a la base el offset de la interrupción para ir a la rutina correspondiente, y se la ejecuta.

Terminamos el ejercicio 2

2.3. Ejercicio 3

Vamos a limpiar el buffer de video para que se vea como lo indica la figura 7. Para ello, creamos la función `imprimir_pantalla` en `screen.c`. La función `imprimir_pantalla` dibuja los bordes (negro), los marcadores (la parte roja y azul de la pantalla correspondiente a cada jugador), los relojes (que luego deberemos actualizar para que respondan a la interrupción que genere una tarea siendo ejecutada), e imprime las vidas. Aquí también se utiliza la función `print_int_sinattr`, que como el nombre lo indica imprime en el lugar buscado del buffer de video el número sin atributos (agarramos los que están en actualizar vidas.) Luego tendremos que hacer esto actualizando las vidas con interrupciones. Cada vez que imprimimos excepto esta usamos la función `print_dada`. Para la medición de los márgenes utilizamos `defines` (archivo `defines.h`)

Ahora vamos a definir la estructura de paginación, y posteriormente, vamos a activar la paginación. Para definir lo primero, se inicializa el manejador de memoria llamando a `mmu_inicializar`, que nos coloca en las posición del principio de las páginas libres (0x10000). A partir de ahora, las páginas serán dadas desde esta dirección de memoria. Luego, inicializamos el directorio del kernel, utilizando para este propósito la función `mmu_inicializar_dir_kernel`.

La función `mmu_inicializar_dir_kernel` realiza lo siguiente:

```
void mmu_inicializar_dir_kernel() {
    unsigned int* pdirectorio = (unsigned int*) ADDR_PAGE_DIR;
    unsigned int* ptabla = (unsigned int*) ADDR_PAGE_TABLE;
    int i, j;
    for (i = 0; i < NUM_TABLES_IDENTITY_MAPPING; i++) {
        pdirectorio[i] = ((unsigned int) ptabla) + PAGE_PRESRW;

        for(j = 0; j < PAGE_SIZE; j++) {
            ptabla[j] = (i * (ENTRIES_TABLE) + j) * PAGE_SIZE + PAGE_PRESRW;
        }
        ptabla += ENTRIES_TABLE;
    }
    for (; i < ENTRIES_TABLE; i++) {
        pdirectorio[i] = 0;
    }
    tlbflush();
}
```

Primero, nos guardamos la direccion del directorio de paginas del kernel(0x27000)en ADDR_PAGE_DIR y la direccion de las tablas de paginas (0x28000) en ADDR_PAGE_TABLE.(ambas definidas en defines.h)

Luego, para el numero de entradas del directorio paginas que requerimos para el identity mapping de nuestro tp (lease, una), se completa las entradas del directorio con la direccion de la tabla de paginas definida anteriormente. Ademas, a cada entrada le disponemos los atributos de estar presente y de poder ser leida y/o escrita(los ultimos bits en 3).

Luego, como necesitamos todas las paginas que nos provee una tabla de paginas,(1024), llenamos cada entrada con la direccion que queremos mapear(0x00000000 a 0x003FFFFFFF), y los atributos de ser leida a la entrada, escrita y presente).

Al terminar lo ultimo, avanzamos ptabla la cantidad de entradas de la pagina.(en este caso no se necesita, pero si se requisiese otra tabla de paginas en el directorio, que el ptabla apunte a la direccion de la proxima tabla.) Luego, se completa las entradas no validas del directorio de paginas con 0 para que no haya problemas.

Por ultimo, como vamos a cambiar la cr3 para inicializar la paginacion, flusheamos la tlb.(para invalidar chache de las traducciones default) Luego movemos la direccion del directorio cargada en cr3, y habilitamos la paginacion seteando el bit 31 de cr0.

Terminamos el ejercicio 3

2.4. Ejercicio 4

En este ejercicio vamos a armar las funciones necesarias para construir la estructura de paginación de una tarea.

Para ello implementamos la función `mmu_inicializar_dir_tarea`, que funciona de la siguiente manera:

```
unsigned int mmu_inicializar_dir_tarea( unsigned char* code, unsigned int dirmapa) {
    unsigned int* pdirectorio = (unsigned int*) mmu_proxima_pagina_fisica_libre();
    int i, j;
    unsigned char* pcodigo_destino = (unsigned char*) DIR_LOG_AFUERA_MEMORIA;
    for (i = 0; i < NUM_TABLES_IDENTITY_MAPPING; i++) {
        unsigned int* ptabla = (unsigned int*) mmu_proxima_pagina_fisica_libre();
        pdirectorio[i] = (unsigned int) ptabla | (unsigned int) PG_PRESENT |
            (unsigned int) PG_WRITE | (unsigned int) PG_KERNEL;

        for(j = 0; j < PAGE_SIZE; j++) {
            ptabla[j] = (i * (ENTRIES_TABLE) + j) * PAGE_SIZE + PAGE_PRESENT;
        }
        ptabla += ENTRIES_TABLE;
    }
    for (; i < ENTRIES_TABLE; i++) {
        pdirectorio[i] = 0;
    }

    mmu_mapear_pagina(DIR_LOG_CODIGO_TAREA, (unsigned int) pdirectorio, dirmapa, PG_USER | PG_WRITE);
    mmu_mapear_pagina(DIR_LOG_PAGINA_TAREA, (unsigned int) pdirectorio, dirmapa, PG_USER | PG_WRITE);

    mmu_mapear_pagina(DIR_LOG_AFUERA_MEMORIA, rcr3(), dirmapa, PG_KERNEL | PG_WRITE);

    for(i = 0; i < PAGE_SIZE; i++) {
        pcodigo_destino[i] = code[i];
    }

    mmu_unmapear_pagina(DIR_LOG_AFUERA_MEMORIA, rcr3());

    tlbflush();
    return (unsigned int) pdirectorio;
}
```

Los parámetros pasados en la función son la dirección donde está el código de la tarea, y la dirección que el jugador le pasa para mapear ese código. Dentro de la función, se declara un puntero llamado `pdirectorio`, al cual se le pasa la dirección de la próxima página física libre (dada por la función `mmu_proxima_pagina_fisica_libre()`). Esta será la dirección del directorio de páginas de la tarea. También se declaran `i` y `j` que usaremos de contadores, y el puntero `pcodigo_destino`, que apunta a la dirección lógica `0x08000000`, donde se ubicará el código de una tarea al ser inicializada (donde se encuentra el código destino).

Luego, mientras `i` sea menor a la cantidad de páginas de identity mapping (que es 1, en este ejercicio), se declara otro puntero `ptabla`, (que también recibe la próxima página libre), que será la dirección de una de las tablas de páginas del directorio de la tarea. En la dirección apuntada por `pdirectorio` en esa posición (`i`) se declara el puntero a la tabla, (las 20 bits + altos). Al declararla, le seteamos los atributos para escribir, estar presente y ser del kernel.

Después, usando la `j` como contador, rellenamos las posiciones de la tabla pasada en esa entrada del directorio con las posiciones donde deberán empezar las páginas (vamos aumentando * el tamaño de página cada vez para que no se solapen, y que la dirección de la primera página de la tabla sea la siguiente a la de la última página de la anterior tabla (de la entrada anterior del directorio de la tarea), de manera que quede toda la paginación en un bloque contiguo de memoria. Le ponemos los atributos de ser presente y poder ser escrita, leída, estar presente y privilegio de supervisor. (sumándole 3)

Completamos el resto de las entradas del directorio de tablas con 0. Ahora ya completadas las paginas, copiamos el codigo a la posicion del mapa que le indica el jugador, y copiamos las paginas a partir de la direccion virtual dada.

Para eso utilizamos la funcion `mmu_mapear_pagina(DIR_LOG_CODIGO_TAREA, (unsigned int) pdirectorio, dirmapa, PG_USER — PG_WRITE)` para mapear la direccion virtual donde se encuentra el codigo de la tarea en la direccion del mapa pasada por el usuario, con los atributos de escribir y ser de usuario. (utilizamos como direccion de cr3 al puntero de directorio de la tarea.). Hacemos lo mismo con la pagina virtual siguiente (que es la pagina mapeada para que la tarea pueda modificarla)

Para copiar el codigo de tareas, se mapea una direccion virtual libre a la direccion del mapa pasado, con permisos de supervisor, y el rcr3 actual, usando `mmu_mapear_pagina(DIR_LOG_CODIGO_TAREA - PAGE_SIZE, rcr3(), dirmapa, PG_KERNEL — PG_WRITE)`

Se copia en la pagina virtual que indica el enunciado el contenido de las pagina de codigo pasado por parametro (lendo desde `i` en 0 hasta el final de la pagina (`tam de pagina-1`)).

Por ultimo se desmapea la pagina auxiliar que fue utilizada para que el kernel copiara a la pagina usando `mmu_unmapear_pagina(DIR_LOG_CODIGO_TAREA - PAGE_SIZE, rcr3())`, donde, `DIR_LOG_CODIGO_TAREA - PAGE_SIZE` es donde habia mapeado la pagina (`0x08000000 - 0x1000`) y `rcr3` el cr3 (puntero).

Se flushea la tlb con `tlbflush` para invalidar la tabla de traducciones, y se devuelve `pdirectorio`, (efectivamente, la direccion que el cr3 deberia usar para ir al directorio de paginas de la tarea).

`mmm_mapear_paginas` recibe de parametros la direccion virtual (`virtual`), el cr3, la direccion fisica (las tres direcciones son pasadas como `unsigned int`), y ademas los atributos que vamos a pasar a la pagina mapeada (como `unsigned char`) y funciona de la siguiente manera:

```
void mmu_mapear_pagina(unsigned int virtual,
unsigned int cr3,
unsigned int fisica,
unsigned char attr) {
    unsigned int *pdirectorio = (unsigned int*) cr3;
    unsigned int *ptabla;
    int i = 0;
    if(!PDE_PRESENT(pdirectorio[PDE_INDEX(virtual)])) {
        ptabla = (unsigned int*) mmu_proxima_pagina_fisica_libre();
        pdirectorio[PDE_INDEX(virtual)] = (unsigned int) ptabla |
        (unsigned int) PG_USER | (unsigned int) PG_WRITE | (unsigned int) PG_PRESENT;
        for(; i < ENTRIES_TABLE; i++)
            ptabla[i] = 0x0;
    }
    else
        ptabla = (unsigned int*) PDE_DIRECCION(pdirectorio[PDE_INDEX(virtual)]);
    if(!PTE_PRESENT(ptabla[PTE_INDEX(virtual)]))
        ptabla[PTE_INDEX(virtual)] = fisica | attr | PG_PRESENT;
    tlbflush();
}
```

Declaramos un puntero `pdirectorio`, al cual le pasamos el `cr3()`, y otro que llamamos `ptabla`. y luego, si la entrada del directorio (apuntada por el `pdirectorio`) de paginas solicitada por el `virtual` (los primeros 10 bits de `virtual`) no esta presente, entonces se pide la direccion de la proxima pagina fisica libre y se la asinga a `ptabla`. A su vez, se coloca `ptabla` en la entrada del `pdirectorio` donde dio que no estaba, (con los atributos de usuario, presente y escribible), y se llena el contenido de la tabla de paginas apuntada por esta `ptabla` de 0.

Si por el contrario esta presente la entrada, `ptabla` le ponemos la direccion que de la entrada, y si no esta presente la entrada de la tabla de paginas pasada en `virtual`, se coloca en esa entrada la direccion de la pagina fisica pasada (efectivamente, poniendola como direccion destino de la paginacion de la direccion virtual dada), junto a los atributos pasados y el atributo de presente.

Por ultimo flusheamos la tlb para invalidar la tabla de cache.

`mmu_unmapear_paginas` recibe de parametros la direccion virtual(`virtual`) y el `cr3` (no recibe los otros dos, porque como la pagina esta mapeada, no los necesita), y funciona de manera similar, ausentando las entradas del directorio donde no hay tablas presentes.

En la funcion se vuelve a declarar un puntero al cual le asignamos el `cr3`, y una `ptabla`. Si la entrada del directorio de paginas apuntado por `pdirectorio` que es pasada por `virtual` esta presente (que es cuando hay que unmapearla), entonces a `ptabla` apunta a la direccion de la tabla de paginas almacenada en esa entrada. Ponemos la entrada de la tabla de paginas apuntada por los bits 21-12 de `virtual` en 0 (no mas presente); y para `i` desde 0 hasta las entradas de la tabla, si no esta presente una paginase sigue, y si no esta presente ninguna pagina en ninguna entrada de la tabla, se pone la entrada en la entrada donde esta la tabla de paginas en 0 (se la unmapea).

Terminamos el ejercicio 4

2.5. Ejercicio 5

En este ejercicio vamos a completar la IDT para que pueda asociar rutinas a la interrupcion de reloj, de teclado y la de software 0x66. Para hacerlo utilizamos la funcion `idt_inicializar` que habiamos usado antes, expandiendo la funcion para ahora incluir a las interrupciones de reloj, de teclado y a la de software 0x66.

Para las interrupciones de reloj y teclado, utilizamos el mismo procedimiento que con las de excepciones, declarando una `GDT_ENTRY`, (con 32 para el reloj y 33 para el teclado), cuyas rutinas son descritas en la `isr` de ese numero (cuya direccion la `GDT_ENTRY` acomoda para que forme el descriptor deseado). Estas dos interrupciones tienen atributos 0 como las anteriores, por lo cual se usa esa funcion.

La interrupcion de software sin embargo, como su nombre lo indica, no es del mismo nivel que las de teclado o reloj, por lo cual utilizamos declaramos una `IDT_ENTRYUSR(102)`, cuya declaracion tiene la misma forma que una `GDT_ENTRY`, con la diferencia de que el nivel de privilegio es 3 en lugar de 0. Su rutina de interrupcion tambien esta en la `isr` de ese numero.

Ahora vamos a describir el funcionamiento de las rutinas de interrupcion de teclado, reloj y 0x66.

La rutina de atencion de reloj consiste en saltar a la funcion `proximo_reloj`, llamar a la funcion `fin_intr_pic1` (que se encarga de avisarle al PIC que se acaba la interrupcion).

`proximo_reloj` tiene este codigo y funciona de la siguiente manera:

```
proximo_reloj:
    pushad
    inc DWORD [isrnumero]
    mov ebx, [isrnumero]
    cmp ebx, 0x4
    jl .ok
    mov DWORD [isrnumero], 0x0
    mov ebx, 0
.ok:
    add ebx, isrClock
    imprimir_texto_mp ebx, 1, 0x0f, 49, 79
    popad
    ret
```

luego incrementa la `DWORD isrnumero` (definida como 0) y lo mueve a `ebx`, y lo compara con 4 (cantidad de posiciones del reloj), si es menor a 4, entonces le sumamos a `ebx` `isrClock` (nos aprovechamos de los tipos para que al sumarle de 0 a 4 nos de uno de los caracteres definido en el string `isrClock`) y lo imprimimos con `imprimir_texto_mp ebx, 1, 0x0f, 49, 79` (para que lo imprima en la parte inferior derecho de la pantalla) y popeas los registros y vuelves. Si es 4, entonces antes de realizar lo anterior limpiamos `ebx` y `isrnumero`, (para volver al principio del reloj, y q).

La rutina de atencion de teclado consiste en salvar los registros, maver al registro al la `KEYBOARD_PORT(0x60)` (lo que viene por el teclado), y luego, ir metiendo uno por uno los caracteres que significan algo para nuestro juego en `ebx` y ver si coincide con al, en cuyo caso saltamos a `tecla_valida`, que mueve `bl` (donde esta el caracter) a `[0xb809e]` (lo que produce que lo muestre en pantalla), y luego (o despues de pasar la comparacion con todos los caracteres validos en caso de no serlo) llama a `fin_intr_pic1`, popea todo y sale.

La rutina de atencion del system 0x66 (`isr(102)`) mueve `eac` a 0x42. (luego, utilizamos esta interrupcion para que la tarea cheque y actualize sus propiedades (SOY, ESTOY, MAPEAR))

2.6. Ejercicio 6

No vamos a cargar las `tss`, en su lugar vamos a hacerlo a mano. Para ello tenemos la funcion `crear_contexto_usr` en el archivo `ctx.asm`.

Para realizar el cambio "a mano" se utiliza una única entrada de la `GDT`, en la que se guardan los únicos datos necesarios para realizar el cambio, que son necesarios guardar en la `TSS` con este modo de manejar el Scheduler, que son el `ESP0` y el `SS0`, o sea los datos correspondientes al stack de nivel cero (kernel). Para ser consistentes, la entrada declarada en la `GDT` posee los atributos de ser de sistema, ser una entrada de interrupt task (no busy al declararla), tener privilegio 0 (kernel) y el tamaño que requerimos (1M)

Para declarar esta entrada usamos una funcion llamada `gdt_inicializar_tss()`, ya que no podemos completar estos valores con la inicializacion estatica, en la que ademas utilizamos un struct llamado `str_gdt_entry2d` para poner los valores, esto es, una estructura que define la entrada en la gdt pero esta solamente separada en los las dos posiciones de memoria que requiere una entrada.(lo y high)

El resto de las entradas de la TSS se guardan en una estructura auxiliar llamada TS que permiten hacer el cambio.

Antes de eso, el kernel actualiza la gdt y luego de setear el correcto funcionamiento de las interrupciones(sin habilitarlas todavia); llama a `sched_idle`, una funcion del scheduler que deja seteado la tarea idle.(mientras no haya cambio, corre la dir de la tarea del ubicada en la `ts_tareas[TS_IDX_IDLE].esp0`(siendo `TS_IDX_IDLE` la ultima entrada de la ts), y el kernel llama a `sched_sched_inicializar`.

Esta funcion, como su nombre lo indica, inicializa el scheduler organiza las tareas, es decir, el quantum de tiempo que recibe cada una, y el orden a correr, asi como tambien se encarga de cambiar las tareas cuando el quantum que este les asigna haya terminado.

```
void sched_inicializar() {
    int i = 0;
    unsigned int ts_idx = TS_START_IDX_SANAS;
    unsigned int x;
    unsigned int y;
    for(; i < MAX_NUM_TAREAS_SANAS; i++, ts_idx++) {
        do {
            y = rand() % ALTO_MAPA;
            x = rand() % ANCHO_MAPA;
        }
        while(!sched_correr_tarea(SCHED_QUEUE_IDX_SANAS, (unsigned char*) DIR_PHY_CODIGO_SANA, x, y));
        screen_mapa_imprimir_tarea_sana(x, y);
    }
}
```

El `sched_inicializar` se fija para todas las sanas si se esta corriendo esa tarea en el mapa, y si no simplemente se ocupa de imprimir las tareas sanas para que se sigan viendo en pantalla. Osea, recorre las tareas que estan definidas en las sanas hasta que alguna interrupcion haga otra cosa, y si no las corre, la imprime.

Contamos con una estructura llamada `queue`, donde estan, tanto para las tareas sanas como para las del jugador a y las del jugador b, la max acantidad que puede haber, corriendo, su lugar en ts, e; de que tipo son, y su tarea en si misma. Con esto podemos ver en cual tarea estamos.

El `sched`, para fijarse si esta corriendo, y para correr una tarea, usa la funcion `sched_correr_tarea`, el cual funciona de la siguiente manera:

```

unsigned int sched_correr_tarea(unsigned int idx_queue, unsigned char* dir_phy_codigo,
unsigned int x, unsigned int y) {
    unsigned int iT = 0;
    unsigned int iQ = 0;
    unsigned int lugar_ocupado = 0;
    queue* q = &run_queues[idx_queue];
    for(; iQ < NUM_QUEUES ; iQ++)
        for(iT = 0; iT < run_queues[iQ].cant; iT++) {
            lugar_ocupado = lugar_ocupado || (run_queues[iQ].tareas[iT].pos_x == x
            && run_queues[iQ].tareas[iT].pos_y == y &&
            run_queues[iQ].tareas[iT].viva == TRUE);
        }

    for(iT = 0; iT < q->cant && q->tareas[iT].viva == TRUE; iT++);
    if(iT < q->cant && !lugar_ocupado)
    {
        q->tareas[iT].ts_idx = q->ts_start_idx + iT;
        q->tareas[iT].pos_x = x;
        q->tareas[iT].pos_y = y;
        crear_contexto_usr(&tareas[q->ts_start_idx + iT], dir_phy_codigo, mmu_dir_mapa(x, y));
        q->tareas[iT].estado_reloj = 0;
        q->tareas[iT].virus = q->jug;
        q->tareas[iT].viva = TRUE;
    }
    return iT < q->cant && !lugar_ocupado;
}

```

Lo que realiza esta función es, poner en q la dirección de la tarea a ejecutar, y luego, fijarse para todas las tareas, donde si indica esta x e y esta la tarea a correr o ocupado el lugar, y si no lo está, se guardan en tareas[it] datos que nos van a importar de esta tarea cuando regresemos, se la posiciona a donde nos vino por parametro, y se llama a crear_contexto_usr que cambia la tarea a la actual, y retorna bool de si hizo eso. (los requerimientos para realizar la acción)

La idea de la función crear_contexto_usuario es en realizar las mismas funciones que realizaría cambio de tss, el de guardar el contexto de una tarea tiene al momento que el scheduler la desaloja, y de cargar el contexto guardado de la próxima tarea a ejecutar por el scheduler, de manera de que la interrupción sea transparente para la tarea.

La función en si misma genera el direccionamiento al mapa y la inicialización de la paginación llamando a mmu_inicializar_dir_tarea, así como a mmu_nueva_pila_kernel, que como dice, genera la pila de kernel que va a necesitar. Carga los registros con la info necesaria:

el ss con donde, en disco de datos de gdt, y los privilegios de usuario, el esp con dir de pila de tarea los flags con los de ctx el cs con el lugar de disco de código de gdt, y provolegops de usuario por ser tarea

- el eip con la dirección lógica de la tarea(código)
- registros generales en 0
- ds con dir de disco de datos de gdt
- ebp0 en dir de la nueva pila de nivel 0 de la tarea
- cr3 con la dirección del directorio de páginas generado
- ebx con dirección de tarea
- esp0 en dir de la nueva pila de nivel 0 de la tarea

dejando así la tarea cambiada.

En el scheduler, para cambiar tareas cuando se deba, se debe poder fijar la siguiente.

Para fijarse la proxima tarea a ejecutar, el sheduler usa la funcion `sched_proxima_tarea()` que funciona de la siguiente manera:

```
unsigned char* sched_proxima_tarea() {
    unsigned char* res = ts_tareas[TS_IDX_IDLE].esp0;
    unsigned int cT = 0, cQ = 0;
    unsigned int iQ;
    unsigned int iT;

    tarea* actual = sched_info_tarea_actual();
    if(actual->viva == TRUE) {
        actual->estado_reloj++;
        actual->estado_reloj %= CANT_ESTADOS_RELOJ;
        screen_actualizar_reloj(current_queue, run_queues[current_queue].tarea_actual, actual->estado_reloj);
    }

    if(!parado) {
        do {
            iQ = (current_queue + cQ + 1) % NUM_QUEUES;
            iT = (run_queues[iQ].tarea_actual + cT + 1) % run_queues[iQ].cant;
            cQ += (++cT) / (run_queues[iQ].cant + 1);
            cT %= run_queues[iQ].cant + 1;

            } while(cQ <= NUM_QUEUES && run_queues[iQ].tareas[iT].viva == FALSE);

        if(cQ <= NUM_QUEUES) {
            current_queue = iQ;
            run_queues[current_queue].tarea_actual = iT;
            res = ts_tareas[run_queues[iQ].tareas[iT].ts_idx].esp0;
            en_idle = 0;
        }
        else {
            en_idle = 1;
        }
    }
    else {
        res = sched_ts_tarea_actual()->esp0;
    }
    return res;
}
```

Que devuelve la dir de la prox tarea. Primero pone en `res` la dir de la tarea anterior que se habia corrido, y define `iT` e `iQ`.

Luego conseguimos la info de la tarea actual, (`sched_info_tarea_actual()`), nos devuelve la cola y la posicion de esta en la cual esta esta tarea, asi como si es idle (en este caso no estaria en las colas) o no). Al tener esta informacion, podemos ver si la tarea sigue viva.

De estar viva, actualizamos su reloj, y luego, mientras el scheduler no este parado, pone a `iQ` e `IT` apuntando a los de la tarea siguiente, y `res` apuntando a al `esp0` de la tarea actual (pila de kernel) y se lo devuelve. En caso de haberse acabado el tiempo, se usa el `iQ` y el `iT` para rellenar la current queneue la tarea actual de la queneue y el resultado a su pila de codigo kernel.

Ademas, si se supero el `NUM_QUEUES` o la tarea muere, entonces se le pone `en_idle=1` para que sheduler corra la tarea idle.

Ademas, en el sheduler mata tareas con la funcion `shed_matar_tarea_actual`, que pone su estado de vivo en falso (en caso de no ser null), pone a correr la tarea idle hasta que se acabe el quorum y actualiza puntajes y relojes. El sheduler tambien se encarga de cambiar los estado de las tareas si estas infectan o so infectadas, y contar la cantidad de infectados de un virus que hay en todo momento.

2.7. Ejercicio 7

En esta seccion vamos a explicar la logica del juego en si y como utilizamos y/o modificamos los archivos para poder modificar esto.

En game.c estan las funciones que tienen que ver con la logica del juego. La mayoría son llamadas por la rutina de atencion de una interrupcion de teclado. Aqui definimos:

game.lanzar, que lanza una tarea. En su codigo game.lanzar, dado jug, averigua cual es su queue(tareas a o b), la dir fisica del codigo de la tarea de jug, y la posicion en donde va a ser lanzada la tarea(cursor). Si el jugador tiene vidas para lanzar tarea, entonces se corre la tarea y se imprime en el mapa la tarea infectadora y se actualizan las vidas de jugador y los puntajes. (llamamos a funciones en screen.c).

game.soy, que, dado el yosoy de una tarea, si la tarea es de jugador a o b, muestra el color que le corresponde por pantalla y actualiza los puntajes y su estado en shed(llama a shed.infectar(yoSoy)).

game.mapear, que obtiene la informacion del virus de la tarea de un jugador, y la posicion del cursor, y a partir de ellos mapeo la pagina a la direccion de pagina correspondiente, (en el medio retiro el cursor para poder mapear la tarea correctamente)

game.obtener_queue_idx_jugador, (switch que me dice cual cola de tareas debo usar, es la llamada por game.lanzar).

game.obtener_dir_phy_codigo_jugador (switch que da la direccion fisica del codigo del jugador)

game.vidas (switch que me da las vidas)

es_jugador (dado jugador, me dice si es a o b)

game.donde, donde esta la tarea actual en la pantalla.

game.mover_cursor (dado jug y dir, hace desplazamientos de cursor segun int, y cambia posicion en la pantalla. La usamos en la interrupcion de teclado)

Como podemos ver, en muchos lugares utilizamos funciones de screen.c. En la ultima funcion descripta, incluso usamos una estructura que definimos en screen.h. La estructura es jugador_visual_jugadores, y guarda el cursor, la cantidad de pag mapeadas, la tarea Infectadora(lugar mapa), la de la ser infectada(lugar mapa), el marcador de vidas y el marcador (reloj) de los jugadores y el tipo de jugador, para ambos jugadores. Ademas tenemos definido el tipo cur, que es un cursor y tiene como atributos pos, visible y abajo(para guardar lo de abajo de un cursor)

Tambien tenemos funciones que imprimen el mapa, tanto sin atributos, sin caracteres, o con ambas.

Tenemos la funcion screen.actualizar_vidas, que para los dos jugadores, ve como es su color de inf, y usa la funcion shed.infectados de antes para obtener los puntos que le corresponden al jugador y los imprime.

La funcion screen.actualizar_reloj, que cambia el estado del reloj por pantalla de la tarea que lo pidio.

La funcion screen.quitar_cursor, que se fija si la posicion de los cursores de los jugadores o el de alguna tarea estan en la misma posicion que la posicion pasado, y si es asi se imprime lo que hay en esa posicion. De manera contraria, se imprime lo que tenia antes la posicion donde esta el cursor.

La funcion screen.ubicar_cursor, que se comporta de la siguiente manera:

```

void screen_ubicar_cursor(cursor* cur, unsigned int x, unsigned int y) {
    int i = 0;
    cursor* cursor_misma_pos = NULL;
    for(; i < CANT_JUGADORES; i++)
        if(visual_jugadores[i].cursor.posicion.x == x && visu
            al_jugadores[i].cursor.posicion.y == y && &visu
            al_jugadores[i].cursor != cur)
            cursor_misma_pos = &visual_jugadores[i].cursor;

    for(; i < MAX_NUM_TAREAS; i++)
        if(cursores_paginas[i].posicion.x == x && cursores_pa
            ginas[i].posicion.y == y && &visual_jugadores[i].cursor != cur)
            cursor_misma_pos = &cursores_paginas[i];

    if(cursor_misma_pos == NULL) {
        cur->abajo.c = screen_mapa_obtener(x, y)->c;
        cur->abajo.a = screen_mapa_obtener(x, y)->a;
        cur->visible.a = screen_mapa_obtener(x, y)->a;
    }
    else {
        cur->abajo.c = cursor_misma_pos->abajo.c;
        cur->abajo.a = cursor_misma_pos->abajo.a;
        cur->visible.a = cursor_misma_pos->visible.a;
    }
    cur->posicion.x = x % ANCHO_MAPA;
    cur->posicion.y = y % ALTO_MAPA;
    screen_mapa_imprimir(&cur->visible, cur->posicion.x, cur->posicion.y);
}

```

Recibe un la direccion de un cursor, y una posicion, y di el cursor de uno de los jugadores esta en el mismo lugar, o si esta en el cursor de alguna de las tareas, entonces le da los mismos atributos al cursor que la del cursor con el cual coincide. Si no coincide con ninguno, entonces guardamos en abajo los atributos de la posicion de abajo de donde voy a poner el cursor, y en visible tambien(pues no ha ninguna tarea o jugador).Por ultimo, le pongo a los atributos de cursor la pos pasada por parametro y lo imprimo.

Tambien hay otras funciones, que guardan la pantalla en una matriz en aparte, screen_visual_jugador, que para un jugador dado, te da sus atributos (de la estructura auxiliar visual jugadores), y screen_mapa-imprimir_tarea_afectora(llama a imprimir con la tarea infectoria de el jug pasado, y ubica los anterior debajo(en cursor)), screen_infectar, que hace lo mismo pero sin los attr de la tarea, y screen_mapa_tarea_sana, que imprime sin caracter la tarea sana en la pos pasada(y ubica cursores debajo)

Con esto el manejo del juego esta estaticamente cerrado, para que funcione, sin embargo, se necesita de que las interrupciones llamen a estas funciones. Debido a eso, actualizamos las rutinas de interrupcion de teclado y la de sistema0x66

- Modificaciones de la interrupcion de teclado:

Cuando se produzca alguna interrupcion que agregue tareas,(las de los shifts), la rutina de interrupcion de teclado de isr llama a game_lanzar. Si se presiona la de las direcciones, se llama a game_mover_cursor(se pasan en ebx, ecx los parametros para llamarla) se mueve 8 para no romper el stack, se llama a fin_intr_pic1 y se vuelve.

- Modificaciones de la interrupcion de reloj:

Se llama a proximo_reloj, luego a fin_intr_pic1 para poder recibir otra interrupcion de ser requerido, luego a shed.ts)tarea_actual y luego de tenerla, con su resultado a sched_proxima_tarea, se compara si son iguales, en cuyo caso se termina la interrupcion, o en caso de no serlo, se llama a cambio_contexto para cambiar el contexto y pasar a la proxima tarea), y se sale.

- Modificaciones de la interrupcion de sistema:

Se salvan los registros y se compara con el valor que significa que la tarea solicita el servicio DONDE, y hace lo mismo con los servicios SOY y MAPEAR, si alguno de ellos coincide, entonces se llama, respectivamente a `game_soy`, `game_mapear` o `game_donde` y luego popea los registros y sale.

Con esto queda bien funcional el juego.

3. Aclaraciones del Makefile/Generales

El Makefile fue modificado de la siguiente manera:

- Agregamos un archivo (`ctx.asm`)
- Agregamos el flag `-shared`, ya que si no producía errores al compilar las tareas.