Resumen

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

Índice

Figura 1: Descripcion de la figura

1. Objetivos generales

El objetivo de este Trabajo Práctico es ...

2. Contexto

Titulo del parrafo Bla bla bla bla. Esto se muestra en la figura 1.

3. Enunciado y solucion

4. Resolución de los Elercicios

4.1. Ejercicio 1

En el ejercicio 1 se completa e inicializa la GDT, con cuatro segmentos, 2 de codigo(uno del kernel y el otro del usuario), y dos de datos(uno del kenrnel y otro del usuario), dirrecionando los primeros 878MB de memoria.(Se dejan las primeras 3 posiciones de la gdt libre, una or ser la nula y las otras dos por reestricciones del tp).El primer indice 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. Alli, 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;
```

Alli, al nulo se le pone todo 0 y a los otros se le pone diferntes parametros dependiedo si es un descriptor de codigo, de datos, o si es un descriptor de sistema. Por ejemplo, el desriptor de codig del kernel que defnimos 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_PRES,                   /* 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

4.1.1. Consideracionesde ej 1

- Se flatea las secciones, poniendoles a todas la misma direccion base (0x00) y de limite se coloca el tamano-1 /0x400 +0x3FF (vamos a poner el de granularidad en 1, para que la cuenta nos de el tamano pedido menos 1)

- El tipo es read/write(0x02) en los de data, y (execute/read) en los de codigo.
- El s es 1 x ser de datos o codigos.
- la dpl es 3 o 0 dependiendo si es un segmento de usuario o del kernel.
- el l esta en 0 x estar en 32 bytes. lel db esta en 1 por el mismo motivo.

  Tambien hay un segmento de video(del kernel) cuya direccion de entrada es la pedida por la catedra(y limite de acuerdo al tamano de la pantalla pedida.)), el resto es como un segmento de codigo 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 desabilitan las interrupciones(cli), se cambia el modo de video(interrupcion)(se va a modo 3h y luego se setea)

  Se habilita la A20 (con una funcion) y se carga la gdt con la funcion lgdt(y la direccion de la tabla(y tam)).

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

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

  Luego se inicializa la pantalla, para ello llamamos a la funcion inicializar_pantalla, que pushea ds, mueve eax al segmento de video,(y a la primera posicion), 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. En

  Terminamos ejercicio 1

4.2. 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 decriptores de interrupciones(desriptores 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 funcion contiene los descriptores que seran puestos en la GDT (lo que hacemos utilizando la instruccion lidt [IDT_DESC], donde IDT_DESK el limite y la base de la IDT.)

Los descriptores de interrupcion estan 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 direccion de la rutina de atencion de interrupciones (definidas como isr(numero de interrupcion)).

Las rutinas de interrupcion consisten en imprimir infinitamente el texto el nombre de la excepcion que el procesador genero, 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 esta el descriptor del segmento donde se encuentran las rutinas de atencion de interrupciones.

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

Terminamos el ejercicio 2

4.3. Ejercicio 3

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

Ahora vamos a definir la estructura de paginacion, y posteriormente, vamos a activar la paginacion. Para definir lo primero, se inicializa el manejador de memoria llamando a mmu_inicializar,que nos coloca en las posicion del principio de las paginas libres(0x10000). A partir de ahora, las paginas seran dadas desde esta direccion de memoria. Luego, inicializamos el directorio del kernel, utilizando para este proposito la funcion mmu_inicializar_dir_kernel.

La funcion 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 0x003FFFFF), 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 direcctorio 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

4.4. Ejercicio 4

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

Para ello implementamos la funcion 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_PRESRW;
}
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 parametros pasados en la funcion son la direccion donde esta el codigo de la tarea, y la direccion que el jugador le pasa para mapear ese codigo. Dentro de la funcion, se declara un puntero llamado pdirectorio, al cual se le pasa la direccion de la proxima pagina fisica libre(dada por la funcion mmu_proxima_pagina_fisica_libre()). Esta sera la direccion del directorio de paginas de la tarea.Tambien se declaran int que usaremos de contadores(i, j) y la el puntero pcodigo_destino, que apunta a la direccion logica 0x08000000, donde se ubicara el codigo de una tarea al ser inicializada(donde se encuentra el codigo destino).

Luego, mientra i sea menor a la cantidad de paginas de identity mapping (que es 1, en este ejercicio), se declara otro puntero ptabla, (que tambien recibe la proxima pagina libre), que sera la direccion de una de las tablas de paginas del directorio de la tarea. En la direccion apuntada por pdirectorio en esa posicion(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.

Despues, usando la j como contador, rellenamos las posiciones de la tabla pasada en esa entrada del directorio con las posiciones donde deberan empezar las paginas(vamos aumentando * el tamano de pagina cada vez para que no se solapen, y que la direccion de la primera pagina de la tabla sea la siguiente a la de la ultima pagina de la anterior tabla (de la entrada anterior del directorio de la tarea), de manera que quede toda la paginacion en un bloque contiguo de memoria. Le ponemos los atributos de ser presente y poder ser escrita, leida, estar presente y privilegio de supervisor.(sumandole 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 mmmu_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(llendo 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 ptablas. A su vez, se coloca ptablas 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, ponienedola 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 mapeda, 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

4.5. 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 sofware 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 sofware 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 descriptas en la isr de ese numero (cuya direccion la GDT_ENTRY acomoda para que formar el descriptor deseado) Estas dos interrupciones tienen atributos 0 como las anteriones, por lo cual se usa esa funcion.

La interrupcion de sofware 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 ibx 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 volves.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 regitro 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))

4.6. 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 tamano 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 tambienque 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 sheduler_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 jugadorb, 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 sheduler, 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(&ts_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 funcion es, poner en q la direccion de la tarea a ejecutar, y luego, fijarse para todas las tareas, donde si donde indica esta x e y esta la tarea a correr o ocupado el lugar, y si no lo esta, 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 accion)

La idea de la funcion crear_contexto_usuario es en realizar las mismas funciones que realizaria cambio de tss, el de guardar el contexto de una tarea tiene al momento que el sheduler la desaloja, y de cargar el contexto guardado de la proxima tarea a ejecutar por el sheduler, de manera de que la interrupcion sea transparente para la tarea.

La funcion en si misma genera el direccionamiento al mapa y la inicializacion de la paginacion llamando a mmu_inicializar_dir_tarea, asi 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 codigo de gdt, y provolegops de usuario por ser tarea

- el eip con la direccion logica de la tarea(codigo)
- 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 direccion del directorio de paginas generado
- ebx con direccion de tarea
- esp0 en dir de la nueva pila de nivel 0 de la tarea

dejando asi la tarea cambiada.

En el sheduler, 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 funciana 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 = (current_queue+1) % NUM_QUEUES;
    unsigned int iT = (run_queues[iQ].tarea_actual+1) % run_queues[iQ].cant;

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

    if(!parado) {
        while(cQ <= NUM_QUEUES && run_queues[iQ].tareas[iT].viva == FALSE) {
            cQ += (++cT) / run_queues[iQ].cant;
            cT %= run_queues[iQ].cant;
            iQ = (current_queue+cQ) % NUM_QUEUES;
            iT = (run_queues[iQ].tarea_actual+1+cT) % run_queues[iQ].cant;

        }
        if(cQ <= NUM_QUEUES) {
            current_queue = iQ;
            run_queues[current_queue].tarea_actual = iT;
            res = ts_tareas[run_queues[current_queue].tareas[run_queues[current_queue].ta
            rea_actual].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 tambien pone en IQ la siguiente cola (de sanas a jugA, de jugA a jugB) y en iT, la tarea siguiente a la ultima que se habia corrido de esa cola.(que es la que se tiene que correr)

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,(se acabo el tiempo) pone a iQ e IT apuntando a los de la tarea actual, 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 curent quenue la tarea_actual de la quenue 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.

se procede a buscar la siguientem moviendo los contadores lo necesario.(Si esta parado, se apunta a la siguiente, sino hace algunas consideraciones de tamano).

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.

4.7. 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.

Cuando se produzca alguna interrupcion que agrege tareas,(las de los shifts), la rutina de interrupcion de teclado de isr llama a game_lanzar, que identifica el jugador y la direccion fisica de donde vino la interrupcion, y el lugar donde estaba el cursor, (llama a funciones auxiliares que guardan/muestran esa informacion) y si

5. Conclusiones y trabajo futuro