Trabajo Práctico 3
System Programming - Pandemia

Organización del Computador 2

1er Cuatrimestre 2013

1. Objetivo

El tercer trabajo práctico de la materia consiste en un conjunto de ejercicios en los que se aplican de forma gradual, los conceptos de *System Programming* vistos en las clases teóricas y prácticas.

En esta ocasión se buscará construir un sistema donde correrá una versión del juego: Infección. Deberán crear el soporte para que los jugadores (tareas) puedan ejecutar las reglas del juego. A su vez, como las tareas son capaces de hacer cualquier cosa, el sistema debe ser capaz de capturar cualquier problema y poder quitar a la tarea del juego.

Los ejercicios que se proponen utilizan los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo enfocados sobre el sistema de protección.

2. Introducción

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

Una computadora al iniciar comienza con la ejecución del POST y el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En este caso dispondremos de un Floppy Disk como dispositivo de booteo. En el primer sector de dicho floppy, se almacena el boot-sector. El BIOS se encarga de copiar a memoria los 512 bytes del sector, a partir de la dirección `0x7c00`. Luego, se comienza a ejecutar el código a partir esta dirección. El boot-sector debe encontrar en el floppy el archivo `kernel.bin` y copiarlo a memoria. Éste último se copia a partir de la dirección `0x1200`, y luego se ejecuta a partir de esa misma dirección. En la figura 1 se presenta el mapa de organización de la memoria utilizada por el kernel.

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

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

- `Makefile` - encargado de compilar y generar el floppy disk.

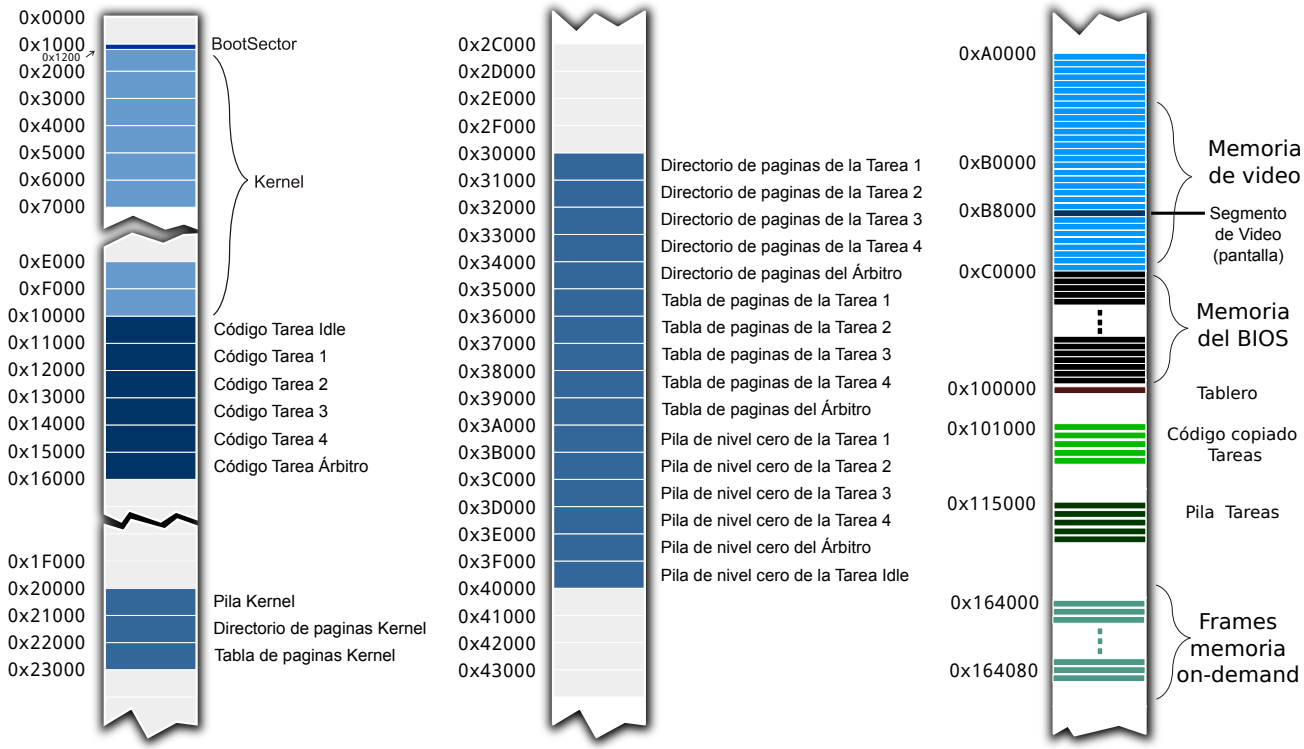


Figura 1: Mapa de la organización de la memoria física del kernel.

- `a20.asm` - rutinas para habilitar y deshabilitar A20.
- `bochsrc` - configuración para inicializar Bochs.
- `defines.h` - constantes y definiciones globales.
- `diskette.img` - la imagen del floppy que contiene el boot-sector preparado para cargar el kernel.
- `game.h` y `game.c` - rutinas asociadas al juego.
- `gdt.h` y `gdt.c` - definición de la tabla de descriptores globales.
- `i386.h` - funciones auxiliares para utilizar assembler desde C.
- `idle.asm` - código de la tarea Idle.
- `idt.h` y `idt.c` - entradas para la IDT y funciones asociadas como `inicializar_idt` para completar entradas en la IDT.
- `imprimir.mac` - macros útiles para imprimir texto por pantalla.
- `isr.h` y `isr.asm` - definiciones de las rutinas para atender interrupciones (Interrupt Service Routines) y la definición de la función `proximo_reloj`.
- `kernel.asm` - esquema básico del código del kernel.
- `mmu.h` y `mmu.c` - rutinas asociadas a la administración de memoria.
- `pic.c` y `pic.h` - funciones `habilitar_pic`, `deshabilitar_pic`, `fin_intr_pic1` y `resetear_pic`.
- `sched.h` y `sched.c` - rutinas asociadas al scheduler.
- `screen.h` y `screen.c` - rutinas para imprimir en pantalla.
- `syscalls.h` - definiciones de las syscalls para que puedan ser llamadas desde las tareas.
- `tss.h` y `tss.c` - definición de entradas de TSS.

- `arbitro.c` - código para la tarea árbitro.
- `tarea1.c` - código para su tarea.
- `tarea2.o`, `tarea3.o`, `tarea4.o` - código de tareas hecha por la cátedra.

A continuación se da paso al enunciado, se recomienda **leerlo en su totalidad** antes de comenzar con los ejercicios. El núcleo de los ejercicios será realizado en clase, dejando cuestiones cosméticas y de informe para el hogar.

3. Infección: El juego

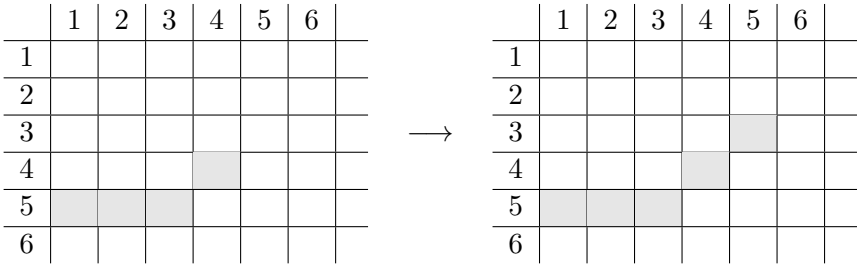
3.1. El juego[1]

Infección es un juego que consiste en crear el mejor virus que infecte un organismo entero. Éste está representado por un tablero de 40×16 células (ancho por alto). Cada virus (un jugador distinto) comienza en una posición definida del tablero y durante el juego se mueve, infectando nuevas células.

El juego termina cuando no quedan más células por infectar. El ganador es el jugador que logra una mayor infección.

Cada jugador por turno puede hacer alguna de las siguientes jugadas:

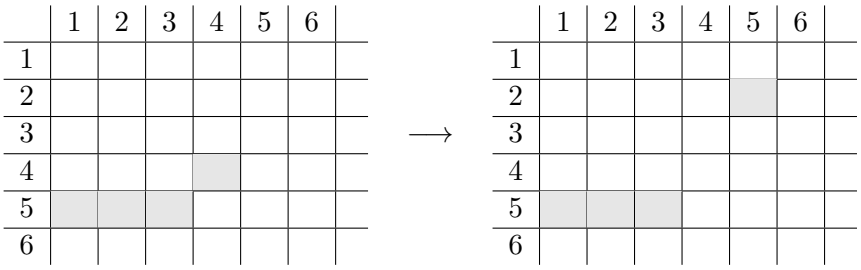
- **Duplicar**: El jugador decide duplicar una célula y lo logra infectando una no infectada adyacente a sí misma. Por ejemplo, dada la siguiente porción del tablero, el jugador gris duplica una de sus células hacia una adyacente llamando a la siguiente función: **duplicar(3, 5)**.



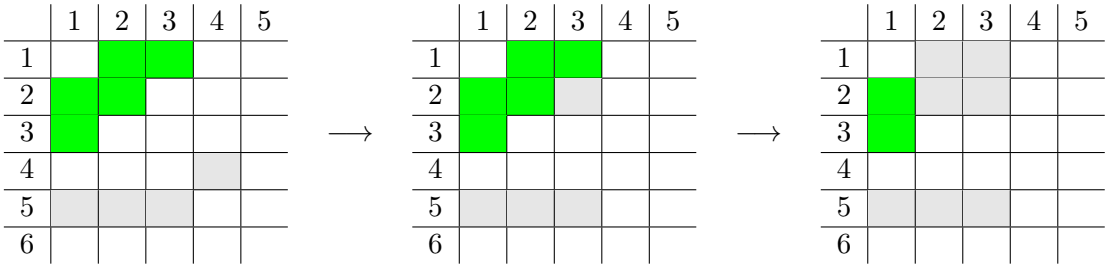
- **Migrar**: El jugador decide cuál célula infectada se mueve y hacia dónde. Una célula puede migrar a una no adyacente a ninguna de sus otras células. Se puede mover como máximo 3 células [2]. Por ejemplo, dada la siguiente porción del tablero, el jugador gris migra una de sus células llamando a la siguiente función: **migrar(4, 4, 2, 5)**

[1] En esta sección sólo se tratan las reglas del juego, y no se tratarán temas implementativos, que sí se verán en las siguientes secciones

[2] Específicamente, la distancia entre dos puntos, es decir, entre $(fila_{inicial}, columna_{inicial})$ y $(fila_{final}, columna_{final})$ es la suma de las diferencias (absolutas) de sus coordenadas y debe ser menor o igual a 3



En cualquiera de los dos casos, las células infectadas por otro virus que queden adyacentes a la nueva célula infectada son ganadas por el último virus (solo las adyacentes). Por ejemplo, para el siguiente estado del juego, cuando se ejecuta el movimiento **migrar(4, 4, 2, 3)** se pasa del primer estado al tercero (el segundo estado nunca sucede realmente, se muestra de modo explicativo).



A continuación se detallará cada una de las partes del sistema que se deben implementar para este trabajo práctico.

3.2. Administrador de Memoria

La memoria física está dividida en una serie de áreas ilustradas en la figura 1.

La primer área incluye al bootloader y al kernel, el cual contiene dentro de su código una copia del código de las tareas. A continuación se encuentran un conjunto de paginas reservadas para las diferentes estructuras que requiere el procesador para administrar la memoria. Estas páginas serán inicializadas por el kernel a medida que se construya el sistema.

La siguiente área está reservada para el BIOS y el video. Estas páginas son mapeadas a dispositivos, por lo que algunas pueden ser modificables y otras no. Por ejemplo, las correspondientes a la pantalla son modificables, para permitirnos escribir en ella. Estas primeras áreas se encuentran dentro del primer megabyte de memoria.

En este caso, el Administrador de Memoria (mmu) implementa una estrategia *On-demand* para cierto espacio virtual. Cada tarea tiene disponible para su uso el siguiente espacio virtual de 32 páginas (i.e. 128KB):

primera pagina direccionable	ultima pagina direccionable
0x003D0000	0x003EF000

Esto significa que cada tarea puede usar, como si ya estuviera mapeada, cualquier dirección perteneciente en ese rango de páginas. En este caso el mmu deberá resolver (atendiendo el

PF) el mapeo necesario. Para realizar este proeza, el mmu cuenta con un espacio de páginas físicas disponibles las cuales irá marcando como ocupadas y se encargara de mapear a medida que cada tarea intente usar una dirección no mapeada. Cada tarea puede/debe usar como máximo 5 paginas distintas. Es trabajo del SO garantizar esto no teniendo que preocuparse por el mal uso, en este aspecto de una tarea.

Las rutinas para la administración de la memoria deben poder inicializar los mapas de memoria para el kernel y cada una de las tareas escribiendo en las páginas indicadas por la figura 1.

El mapa de memoria que se debe construir para cada tarea se puede ver en la figura 2.

Es importante destacar que en este caso, a modo de simplificación las tareas no devolverán memoria pedida dado que el SO no lo soportará.

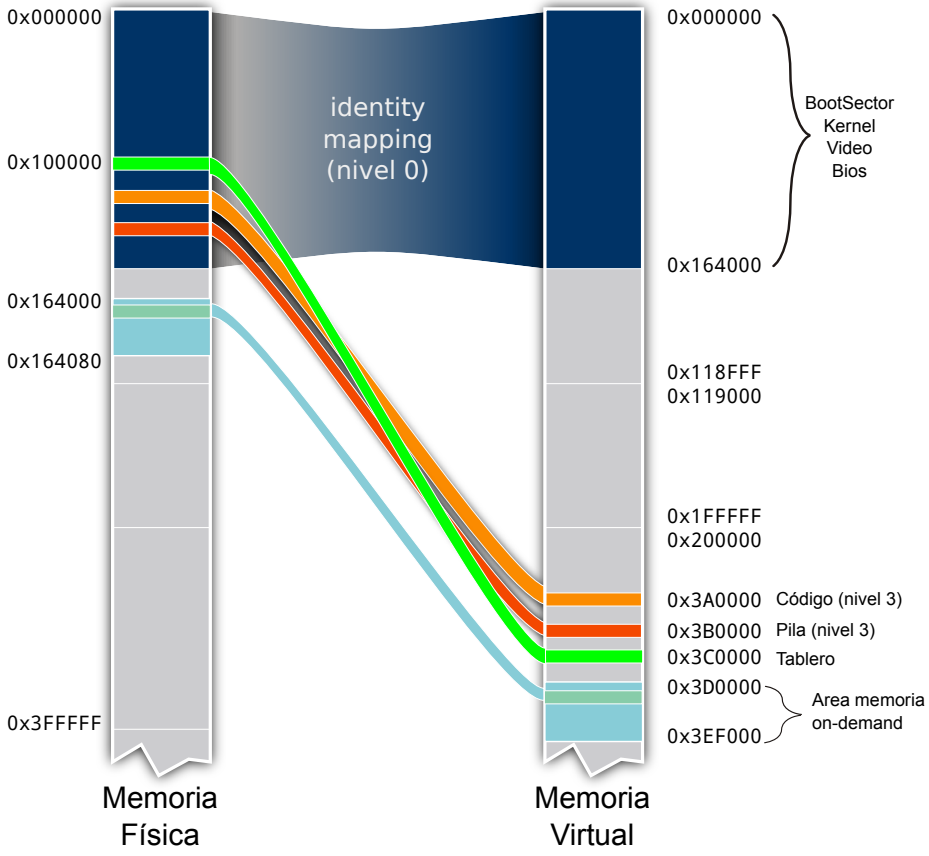


Figura 2: Mapa de memoria de un tarea

3.3. Servicios del "OS"

3.3.1. Servicios para tareas de jugadores

El sistema operativo provee mediante interrupciones los dos servicios mencionados anteriormente. Todos los servicios referidos al juego entran por las interrupciones `0x80`. y `0x90`

Los parámetros de los servicios se pasan por registro, esto quiere decir que se setea el valor de determinados registros y luego se llama al servicio. En la siguiente tabla se indica su uso.

	Servicios	
registros	`duplicar`	`migrar`
`eax` (in)	111	222
`ebx` (in)	*fila*	*fila_src*
`ecx` (in)	*columna*	*columna_src*
`edx` (in)	-	*fila_dst*
`esi` (in)	-	*columna_dst*

El valor de retorno es devuelto por el registro **eax**. En caso que la acción se haya podido concretar el resultado en eax será **1**, en caso contrario **0**.

El tablero está representado en memoria como una matriz del tamaño ya mencionado en una posición prefijada y conocida por las tareas. Cada celda de la matriz es un número de 8 bits sin signo. Las celdas libres (aquellas que no han sido infectadas por ningún jugador) están representadas por el valor `0xFF`. Las que sí están infectadas, tienen el número de jugador que la infectó (es decir, de `1` a `4`). Las escrituras sobre la matriz se realizan solo a través de los servicios queda brinda el "OS" mientras que la lectura está permitida para cualquier tarea.

El tablero se indexa desde cero, es decir, `fila` está dentro del rango $[0, 16)$ y `columna` dentro del rango $[0, 40)$.

3.3.2. Servicios para tarea árbitro

El sistema operativo debe proveer a la tarea árbitro los siguientes servicios:

- Terminar juego: Este servicio da por terminado el juego sacando a todas las tareas del scheduler y poniendo a correr la tarea Idle. Para ello, establece el flag `game_finalizado` en `True`. A partir de este flag el scheduler sabrá que es momento de pasar a ejecutar la tarea Idle.
- Iniciar juego: Este servicio inicializa el tablero, estableciendo las posiciones iniciales para cada jugador (están definidas en `game.h`).

Todos estos servicios son usados mediante la interrupción `0x90`. La interfáz de los mismos es la siguiente:

	Servicios	
registros	`terminar`	`iniciar`
`eax` (in)	200	300

3.4. Tareas

En el SO correrán 7 tareas:

- 4 correspondientes a cada jugador en anillo 3.
- La tarea árbitro en anillo 2.
- La tarea idle que solo correrá cuando no haya un juego activo, en anillo 0.
- La tarea inicial tendrá como única función permitir saltar al inicio del juego (tarea árbitro).

3.5. Tarea árbitro

El árbitro es una tarea especial. Corre en anillo 2 y es la encargada de determinar si el juego terminó y de graficar siempre el tablero (tiene acceso de lectura y escritura a la memoria de video). El pseudocódigo de esta tarea es el siguiente:

```
syscall_iniciar()

while ( !juego_terminado() ) {
  calcular_puntajes()
  actualizar_pantalla()
}

calcular_puntajes()
actualizar_pantalla()
imprimir_ganador()

syscall_terminar()
```

donde `juego_terminado()` es una función que evalúa el tablero actual y decide si el juego terminó. El juego termina cuando no quedan más células por infectar. En el caso que el juego haya terminado se debe mostrar el ganador en pantalla y, luego, usar el servicio del SO *terminar* (descripto en 3.3). Si no hay un ganador (es decir, el juego sigue) se deberá graficar el estado del juego.

El tablero deberá ser mostrado en pantalla como una matriz de 40×16. Cada célula infectada por un jugador deberá pintarse del color que distingue a ese jugador. Cada célula libre deberá ser negra. A su vez, se deberá tener un marcador con la cantidad de células infectadas que tiene cada jugador con el número y el color de cada jugador.

3.6. Scheduler

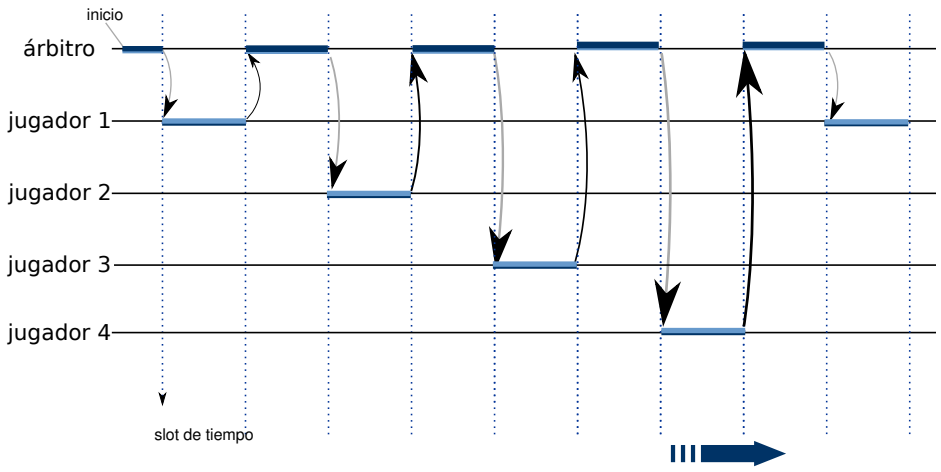


Figura 3: Ejemplo de funcionamiento del *Scheduler*

El scheduler es el encargado de simular los turnos del juego. Para eso se define un *quantum*. El *quantum* es la cantidad de timer ticks que va a correr cada tarea antes de perder el turno. Las tareas **también pierden el turno** luego de hacer una movida exitosa (i.e. luego de ejecutar alguna de las acciones (ver 3.1))

Inicialmente la primera tarea en correr es el árbitro, como se ve en la figura 3. Luego, cuando este gasta su quantum se lanza la tarea del jugador 1. Esta corre hasta que se termine su tiempo o hasta que realiza una acción exitosa, luego debe correr la tarea árbitro nuevamente. Después de transcurrido otro slot de tiempo pasa a correr la tarea del jugador 2. Así sucesivamente, alternándose las tareas de los jugadores y el árbitro.

3.7. Pause

El sistema operativo deberá pausar el juego cuando se toque la tecla **P** (pausar). En dicho caso se dejará terminar de ejecutar la tarea actual hasta que termine su slot de tiempo para luego ejecutarse la tarea Idle. Notar que si el graficador no se ejecuta (pues podría no ser la tarea donde corriendo al momento de hacer pausa) la pantalla puede no estar actualizada. No se correrá ninguna otra tarea hasta que se presione la tecla **R** (reanudar) la cuál reanuda el juego.

3.8. Visualización

La visualización del juego es importante. A modo de resumen, a continuación se presentan aspectos que no pueden faltar:

- Jugadores: Cada jugador tiene un color distinto.
- El tablero de juego: Cada célula no infectada tiene fondo negro. Cada célula infectada lleva el color del jugador que la infectó.
- Tabla de puntaje: Al lado del tablero deberá presentarse una tabla de puntajes. Esta deberá contener 4 filas que indiquen cuántas células infectadas tiene cada jugador (el color de fondo de cada fila, corresponde al jugador en cuestión). En el caso de que el juego esté terminado, deberá haber una etiqueta indicando el ganador (o ganadores, en caso de empate).
- Bloque de estados: Este deberá contener una línea por tarea. Comienza con el reloj de cada tarea, el cual debe girar siempre que la tarea este ejecutándose. A su vez deberá contener el número de jugador con el fondo correspondiente a su color. Seguido de esto, en el caso que corresponda, se expresará el mensaje de error por el cual la tarea no sigue en juego (por ejemplo, la tarea fue sacada del juego por *Page Fault*, *Error de Protección General*, etc).
- Línea de estado final: La última línea deberá contar con el reloj de la tarea idle que solo debe correr cuando esta esté corriendo. Y por último, deberá contener el reloj general del juego, este deberá correr en cada timer tick.

Los siguientes screenshots muestran dos instancias de juego de un diseño posible que cumple con esta descripción. Habrá premio para los diseños que más se destaquen.

Figura 4: Ejemplos visualización. Izquierda: Screenshot del juego (notar que la tarea Roja cometió un error de visión por cero). Derecha: Screenshot del juego terminado. Ganador Jugador Cyan (notar el reloj de la tarea Idle corriendo)

4. Ejercicios

4.1. Ejercicio 1

a) Completar la Tabla de Descriptores Globales (`GDT`) con 6 segmentos, dos para código de nivel 0, 2 y 3; y otros dos para datos de nivel 0, 2 y 3. Estos segmentos deben direccionar los primeros 2GB de memoria.

b) Completar el código necesario para pasar a modo protegido y setear la pila del kernel en la dirección 0x20000.

c) Declarar un segmento adicional que describa el área de la pantalla en memoria.

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar la primer y última línea de color de fondo negro y letras blancas. Para este ejercicio se debe escribir en la pantalla **usando el segmento** declarado en el punto anterior (para los próximos ejercicios se accederá a la memoria de video por medio del **segmento de datos** de 2GB).

> **Pregunta 1:** ¿Qué ocurre si se intenta escribir en la fila 26, columna 1 de la matriz de video, utilizando el segmento de la `GDT` que direcciona a la memoria de video? ¿Por qué?

> **Pregunta 2:** ¿Qué ocurre si no se setean todos los registros de segmento al entrar en modo protegido? ¿Es necesario setearlos todos? ¿Por qué?

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

4.2. Ejercicio 2

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

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

Pregunta 3: ¿Cómo se puede hacer para generar una excepción sin utilizar la instrucción `int`? Mencionar al menos 3 formas posibles.

Pregunta 4: ¿Cuáles son los valores del stack cuando se genera una interrupción? ¿Qué significan? (Indicar para el caso de operar en nivel 3 y nivel 0.

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

4.3. Ejercicio 3

a) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el Kernel (`mmu_inicializar_dir_kernel`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x163FFF`, como ilustra la figura 2. Además, esta función debe inicializar el directorio de páginas en la dirección `0x21000` y la tabla de páginas en `0x22000`.

b) Completar el código necesario para activar paginación.

c) Escribir una rutina que imprima el nombre del grupo en pantalla. Debe estar ubicado en la primer línea de la pantalla.

Pregunta 5: ¿Puede el directorio de páginas estar en cualquier posición arbitraria de memoria?

Pregunta 6: ¿Es posible acceder a una página de nivel de kernel desde usuario?

Pregunta 7: ¿Se puede mapear una página física desde dos direcciones virtuales distintas, de manera tal que una esté mapeada con nivel de usuario y la otra a nivel de kernel? De ser posible, ¿Qué problemas puede traer?

4.4. Ejercicio 4

a) Escribir una rutina (`mmu_inicializar`) que se encargue de inicializar las estructuras necesarias y limpiar el mapa de memoria en pantalla.

b) Escribir una rutina (`mmu_inicializar_tarea_jugador`) encargada de inicializar un directorio de páginas y tabla de páginas para una tarea jugador. Respetando la figura 2. La rutina debe copiar el código de la tarea a su área asignada y mapear dicha página a la dirección virtual `0x3A0000`. Además debe mapear la página de pila correspondiente a la tarea a la dirección virtual `0x3B0000`. A su vez, debe mapear la dirección física `0x00100000` a la dirección virtual `0x003C0000` como solo lectura. De esta manera las tareas pueden consultar el tablero para conocer su estado y tomar decisiones.

c) Escribir una rutina (`mmu_inicializar_tarea_arbitro`) encargada de inicializar un directorio de páginas y tabla de páginas para la tarea árbitro. Debe realizar los mapeos análogos al punto anterior pero además debe mapear con identity mapping la memoria de video (`0xB8000`) con permisos de lectura y escritura.

d) Escribir la función:

I- `mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned int attrs)`

Permite mapear la página física correspondiente a `fisica` en la dirección virtual `virtual` utilizando `cr3` con los atributos `attrs`.

e) Construir un mapa de memoria para tareas e intercambiarlo con el del kernel, luego cambiar el color del fondo del primer carácter de la pantalla y volver a la normalidad. La finalidad de esto es testear que el mapeo creado por las funciones anteriores sea válido.

Pregunta 8: ¿Qué permisos pueden tener las tablas y directorios de páginas? ¿Cuáles son los permisos efectivos de una dirección de memoria según los permisos del directorio y tablas de páginas?

Pregunta 9: ¿Es posible desde dos directorios de página, referenciar a una misma tabla de páginas?

Pregunta 10: ¿Que es el TLB (Translation Lookaside Buffer) y para qué sirve?

Nota: Por la construcción del kernel, las direcciones de los los mapas de memoria (`page directory` y `page table`) están mapeadas con *identity mapping*. En los ejercicios en donde se modifica el directorio o tabla de páginas, hay que llamar a la función `tlbflush` para que se invalide la cache de traducción de direcciones. Tener en cuenta que no se va a utilizar más de 4MB de memoria.

4.5. Ejercicio 5

a) Completar las entradas necesarias en la `IDT` para asociar una rutina a la interrupción del reloj, otra a la interrupción de teclado y por último a las que corresponden a las interrupciones de servicio: `0x80` y `0x90`. A su vez, se deberá cambiar el handler de las demás interrupciones de modo que si una tarea de un jugador genera alguna excepción esta sea eliminada del juego (i.e. del scheduler). Las células infectadas por el jugador de la tarea deben quedar como están.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada tick llame a la función `pantalla_rejoj`. La misma se encarga de mostrar, por cada vez que se llama, la animación de un cursor rotando (ver figura de sección 3.8). A su vez también deberá mover el cursor de la tarea que esta corriendo. La función `pantalla_rejoj` está definida en `isr.asm`.

Pregunta 11: ¿Qué pasa si en la interrupción de teclado no se lee la tecla presionada?

Pregunta 12: ¿Qué pasa si no se resetea el PIC?

4.6. Ejercicio 6

a) Definir 7 entradas en la `GDT`. Una reservada para la `tarea_inicial`, otra para la tarea `IDLE` y las 5 restantes para cada una de las tareas que se van a ejecutar en el sistema.

b) Completar la entrada de la `TSS` correspondiente a la tarea `IDLE`. Esta información se encuentra en el archivo `tss.c`. La tarea `IDLE` se encuentra en la dirección `0x00010000`. La pila se alojará en la página `0x0003F000` y será mapeada con *identity mapping*. Esta tarea está compilada para ser ejecutada desde la dirección `0x3A0000` y la misma debe compartir el mismo `CR3` que el kernel.

c) Completar el resto de las entradas del arreglo de las `TSS` definidas con los valores correspondientes a las tareas que correrán en el sistema. El código de las tareas se encuentra, inicialmente, a partir de las direcciones 0x00011000 ocupando una pagina de 4K para cada tarea. Estos deben ser copiados a partir de la dirección física 0x00101000 y debe ser mapeado para cada tarea a la dirección virtual 0x3A0000. Para las paginas de las pilas de anillo 3, el espacio físico dedicado a estas empieza en 0x00115000 y debe ser mapeado, para toda tarea a la dirección 0x003B0000[3].Para el mapa de memoria se debe construir uno nuevo para cada tarea utilizando la función `mmu_inicializar_tarea_jugador`.

d) Completar la entrada de la `GDT` correspondiente a la `tarea_inicial`.

e) Completar la entrada de la `GDT` correspondiente a la tarea `IDLE`.

f) Completar el resto de las entradas de la `GDT` para cada una de las entradas del arreglo de `TSS`s de las 5 tareas que se ejecutarán en el sistema.

[3]Para una referencia más cómoda de todas las direcciones, mirar el archivo `defines.h`

g) Escribir el código necesario para ejecutar la tarea `IDLE`, es decir, saltar intercambiando las `TSS`, entre la `tarea_inicial` y la tarea `IDLE`.

Pregunta 13: Colocando un breakpoint luego de la cargar una tarea, ¿cómo se puede verificar, utilizando el debugger de Bochs, que la tarea se cargó correctamente? ¿Cómo se llega a esta conclusión?

Pregunta 14: ¿Cómo se puede verificar si la conmutación de tarea fue exitosa?

Pregunta 15: Se sabe que las tareas llaman a la interrupción `0x80` y por `0x90`, ¿Qué ocurre si está no esta implementada? ¿Por qué?

Nota: En `tss.c` está definido un arreglo llamado `tss` que contiene las estructuras `TSS`. Trabajar en `tss.c` y `kernel.asm`.

4.7. Ejercicio 7

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

b) Crear la función `sched_proximo_indice()` que devuelve el índice en la `GDT` de la próxima tarea a ser ejecutada. Está debe tener en cuenta lo descripto en la sección 3.6.

c) Modificar la rutina de la interrupción `0x80` y `0x90`, para que implemente los servicios del sistema según se indica en la sección 3.3.

d) Modificar el código necesario para que se realice el intercambio de tareas por cada tick de reloj (si corresponde). El intercambio se realizará por la tarea que indique la función `sched_proximo_indice()`. Es decir, si la tarea actual consumió todo su slot de tiempo (la cantidad de timer ticks que corre cada tarea por turno) debe cambiar sino, debe seguir corriendo. Esta funcionalidad la da la función mencionada anteriormente.

e) Modificar las rutinas de excepciones del procesador para que impriman el problema que se produjo en pantalla, desalojen a la tarea que estaba corriendo y corran la próxima, indicando en pantalla por qué razón fue desalojada la tarea en cuestión.

4.8. Ejercicio 8 (optativo)

a) Crear una tarea propia con *inteligencia artificial* que luche a muerte contra las tareas creadas por los docentes.

- No ocupar más de 4kb.
- Tener como punto de entrada el cero.
- Estar compilada para correr desde la dirección `0x3A0000`.
- Utilizar sólo los servicios presentados en el trabajo práctico.

b) Explicar en pocas palabras qué estrategia utilizaron en la tarea luchadora del ítem anterior en términos.

c) Si consideran que su tarea es capaz de enfrentarse contra las tareas del resto de sus compañeros, pueden enviar el **binario** de la tarea a la lista de docentes indicando los siguientes datos:

   - Nombre de la tarea luchadora, *ej: "La Topadora"*.
   - Características letales, *ej: Es precavida, no siempre ataca.*
   - Sistema de defensa, *ej: La mejor defensa es un buen ataque.*

   Nota: Para los primeros puestos habrá premios sorpresa el día de la competencia.

5. Entrega

La resolución de los ejercicios se debe realizar gradualmente. Dentro del archivo `kernel.asm` se encuentran comentarios (que muestran las funcionalidades que deben implementarse) para resolver cada ejercicio. También deberán completarse el resto de los archivos según corresponda.

A diferencia de los trabajos prácticos anteriores, en este trabajo está permitido modificar cualquiera de los archivos proporcionados por la cátedra, o incluso tomar libertades a la hora de implementar la solución; siempre que se resuelva el ejercicio y cumpla con el enunciado.

Parte de código con el que trabajen está programado en ASM y parte en C, decidir qué se utilizará para desarrollar la solución es parte del trabajo.

Se deberá entregar un informe que describa **detalladamente** la implementación de cada uno de los fragmentos de código que fueron construidos para completar el kernel. En el caso que se requiera código adicional también debe estar descripto en el informe. Cualquier cambio en el código que proporcionamos también deberá estar documentado. Se deberán utilizar tanto pseudocódigos como esquemas gráficos, o cualquier otro recurso pertinente que permita explicar la resolución. Además se deberá entregar en soporte digital el código e informe; incluyendo todos los archivos que hagan falta para compilar y correr el trabajo en Bochs.

También se deberá incluir una sección con todas las respuestas a las preguntas. Si es necesario, pueden incluirse capturas de pantalla, pseudocódigos y diagramas.

La fecha de entrega de este trabajo es **18-6-2013** y deberá ser entregado a través de la página web en un solo archivo comprimido en formato `tar.gz`, con un límite en tamaño de 10Mb. El sistema sólo aceptará entregas de trabajos hasta las **16:59** del día de entrega.

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