Trabajo Práctico 3
System Programming - Mondrian

Organización del Computador 2

Primer Cuatrimestre 2012

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. Se busca construir un sistema minimal que permita correr varias tareas concurrentemente, y administrar la memoria; conservando información sobre qué tarea usa cada parte de la misma. Los ejercicios utilizan los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo.

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 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 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.

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

- `Makefile` - encargado de compilar y generar el floppy disk.
- `bochsrc` - configuración para inicializar Bochs.
- `diskette.img` - la imagen del floppy que contiene el boot-sector preparado para cargar el kernel.
- `kernel.asm` - esquema básico del código para el kernel.
- `gdt.h` y `gdt.c` - esquema de código donde definir la tabla de descriptores globales.

- `tss.h` y `tss.c` - esquema de código donde definir entradas de TSS.
- `idt.h` y `idt.c` - esquema de código donde definir entradas para la IDT y funciones asociadas como `inicializar_idt` para completar entradas en la IDT.
- `isr.h` y `isr.asm` - definiciones de las rutinas para atender interrupciones (Interrupt Service Routines) y la definición de la función `proximo_reloj`.
- `sched.h` y `sched.c` - esquema de código donde definir las rutinas asociadas al scheduler.
- `mmu.h` y `mmu.c` - esquema de código donde definir las rutinas asociadas a la administración de memoria y buffer para imprimir en pantalla.
- `A20.asm` - rutinas para habilitar y deshabilitar A20.
- `macrosmodoprotegido.mac` y `macrosmodoreal.mac` - macros útiles para imprimir por pantalla y transformar valores.
- `tareas.tsk` - binario de las tareas.
- `i386.h` - funciones auxiliares para utilizar assembly desde C.
- `pic.c` y `pic.h` - funciones `habilitar_pic`, `deshabilitar_pic`, `fin_intr_pic1` y `resetear_pic`.

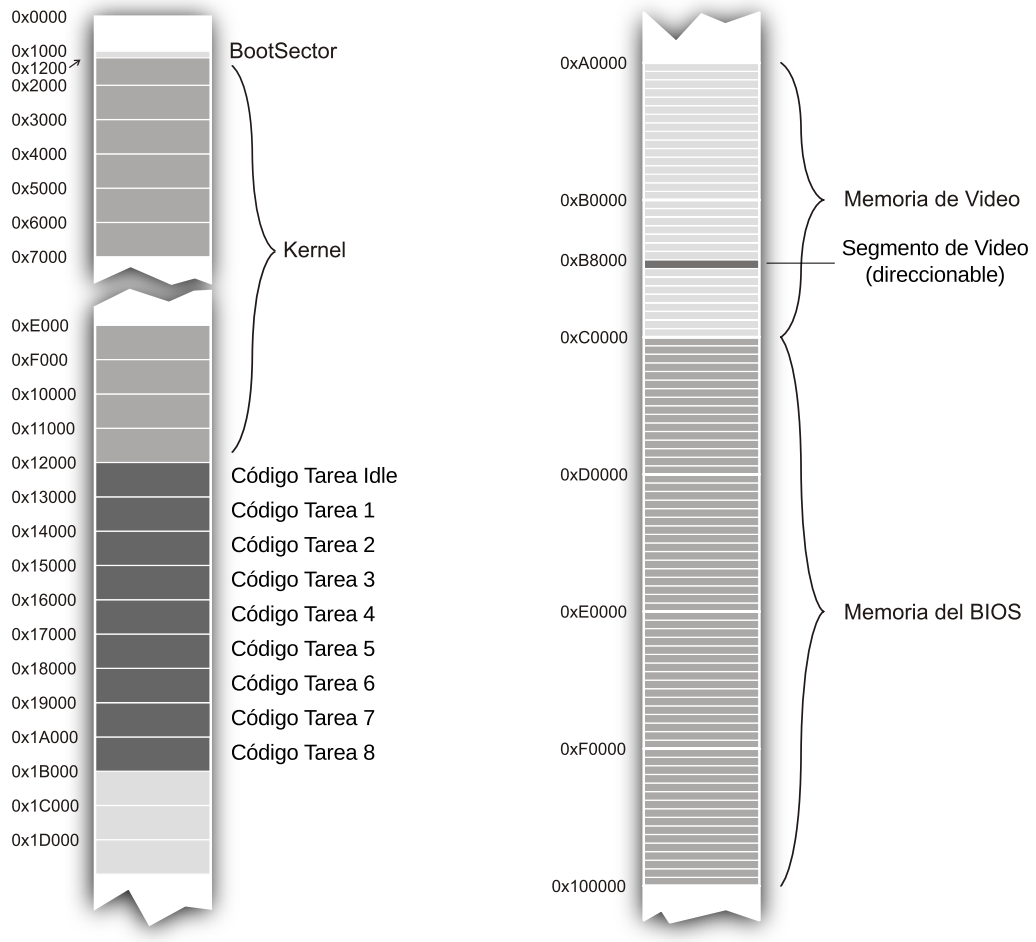


Figura 1: Mapa de la organización de la memoria física del kernel.

3. El orgullo de Piet Mondrian

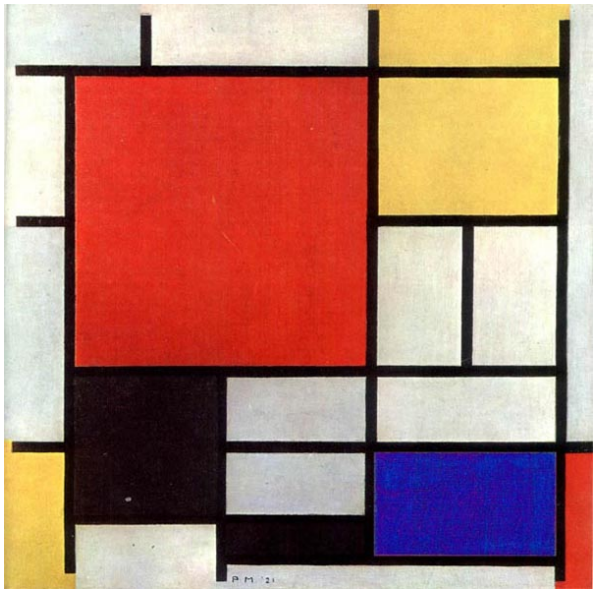


Figura 2: Composición Piet Mondrian. 1921

Piet Mondrian fue un pintor vanguardista holandés y uno de los fundadores del neoplasticismo. Un ejemplo de su arte se puede ver en la figura 2. Cuentan algunas historias que este pintor esperaba por horas a que sus cuadros cambien mágicamente de color, cosa que nunca sucedía. En este trabajo practico vamos a hacer posible este sueño. Utilizando la pantalla de nuestro computador de confianza vamos a mostrar cuadraditos de colores que cambien todo el tiempo.

Como esta tarea resulta muy compleja (pintar cuadraditos en pantalla no es una joda), se nos ocurrió hacer que los cuadraditos de colores representen páginas de memoria pedidas por un proceso.

Para esto, se va a contar con un *Administrador de Memoria* (MMU) que pueda responder a los pedidos de memoria de las tareas. Las tareas o procesos van a tener la capacidad de solicitar una nueva página de memoria al sistema, y éste se encargará de configurar todo lo necesario para satisfacer la solicitud.

En el sistema habrá más de una tarea corriendo, cualquiera de las cuales puede solicitar una página de memoria. Estas tareas, que van a correr concurrentemente, van a ser controladas por un *Scheduler*. Éste es el encargado de dar el control a una tarea por un tiempo (llamado *quantum*), y de intercambiar la tarea actual con la siguiente en la lista de tareas por correr.

A continuación se detallan las características de las tareas, el administrador de memoria y el scheduler.

3.1. Administrador de Memoria Lenteja

La memoria, por medio del sistema de paginación, es dividida en páginas. Éstas pueden ser mapeadas a una tarea que esté corriendo en la computadora. Para controlar este meca-

nismo, se debe implementar un "administrador de memoria lenteja", al cual se le encargan 3 responsabilidades.

La primera es la de manejar el pedido de una página nueva. Para esto, debe buscar en las páginas libres del sistema alguna página que se pueda mapear a la tarea que realizó la solicitud.

La segunda responsabilidad es la de retornar una página pedida. Para esto, debe buscar la dirección física donde se encuentra la página pedida, y luego marcarla como libre; tanto en la estructura del administrador de memoria como en las tablas de la tarea.

La última responsabilidad, y la más importante para Piet Mondrian, es la de pintar cuadraditos de colores en pantalla. Para lograrlo, algunos caracteres de la pantalla van a representar las páginas que fueron usadas o están libres. Por medio de un color de fondo y un número, se identificará qué proceso tiene cada página.

3.2. Tareas Inconformistas

Las tareas o procesos pueden interactuar con el sistema de dos formas, imprimiendo en pantalla, pidiendo una nueva página de memoria; o devolviendo una página que pidieron anteriormente.

Para solicitar una nueva página de memoria se utiliza un llamado al sistema en donde se indica la dirección de memoria virtual donde esta página debe ser mapeada. Cuando dicha página quiere ser liberada se utiliza otro mecanismo del sistema, que; dada una dirección virtual donde está mapeada alguna página; retorna ésta al conjunto de páginas libres. El sistema debe controlar qué páginas son utilizadas y por qué procesos. Además, las tareas pueden escribir en pantalla. Para esto, cuentan con un llamado al sistema que imprime en pantalla un buffer de 20 caracteres pasado por la tarea.

Lamentablemente, las tareas no están muy bien programadas y pueden tener la mala idea de acceder fuera de su espacio de memoria. Se debe poder capturar este error y quitar a la tarea del sistema.

3.3. Topo Yiyo Scheduler

El Topo Yiyo es un personaje infantil de valores muy respetables, por esta razón fue elegido como el representante para repartir tiempos a los procesos. En este sistema los procesos se van a ejecutar de a uno por vez en forma cíclica durante una determinada cantidad de *ticks* de reloj. Esta cantidad puede ser variable y la vamos a denominar *quantum*. El *scheduler* debe tener control de cada uno de los procesos que se están ejecutando, ya que debe identificar el siguiente proceso a ejecutar. Además debe controlar cual es el *quantum* que le corresponde a cada proceso.

4. Implementación

En esta sección se trataran los detalles técnicos que hacen a la implementación del trabajo práctico. Se deberá implementar todo lo mencionado anteriormente con la excepción de las tareas; las cuales son provistas por la cátedra en formato binario listo para ejecutar. Se debe respectar estrictamente cada detalle mencionado, ya que de lo contrario el sistema no va a funcionar correctamente.

El sistema ejecutará 8 tareas de forma concurrente, las cuales utilizarán compulsivamente todos los servicios provistos por el propio sistema. Estas ocho tareas serán cargadas de forma estática y tendrán asignada una porción de la pantalla donde podrá ser escrito su buffer de caracteres.

La pantalla se dividirá de la siguiente forma:

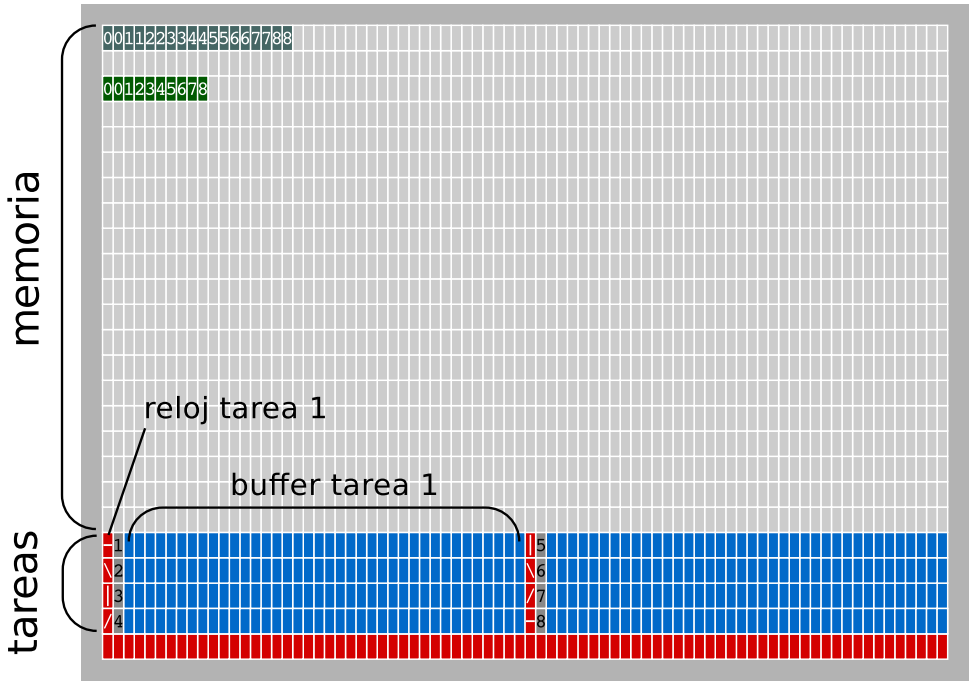


Figura 3: Forma en que la pantalla debe ser dividida

4.1. Servicios del "OS"

El "OS" provee tres servicios a las tareas, el primero para pedir páginas, el siguiente para liberarlas y el último para escribir en pantalla. Todos estos servicios están agrupado bajo la interrupción número 36, dentro del código de la misma se debe seleccionar qué servicio se quiere utilizar.

Los parámetros de los servicios se pasan por registro respetando la siguiente tabla:

Servicio	`malloc_page`	`free_page`	`print_buffer`
`eax`	101	202	303
`ebx`	*dir. virtual*	*dir. virtual*	*dir. buffer*

4.2. Mapa de memoria en pantalla

En la figura 3 se puede ver la sección superior de la pantalla de 80×20 en donde se imprimirán las páginas de memoria utilizadas. Este mapa comenzará en la dirección `0x100000` de memoria física. Cada 4kb corresponderán a una nueva página de memoria. Esto ocurre hasta completar las 1600 páginas que caben en la pantalla.

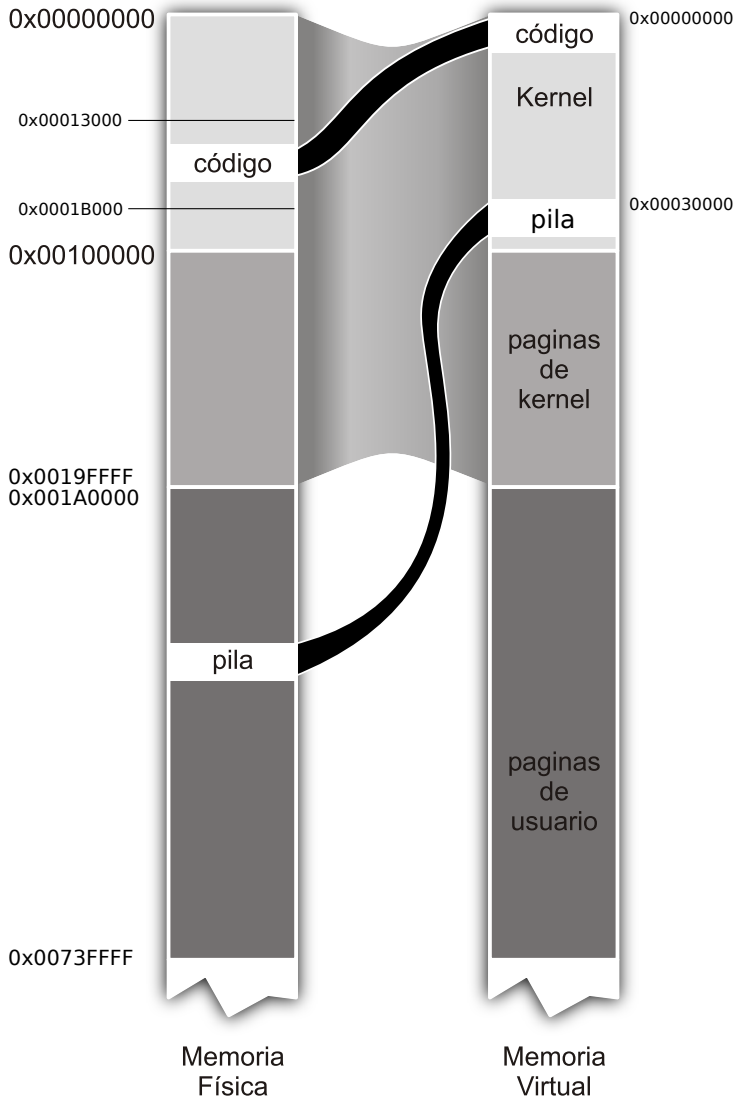


Figura 4: Mapa de la organización de la memoria virtual para cada tarea

Las páginas del mapa serán divididas en dos secciones, “páginas del kernel” (las primeras 160 páginas) y “páginas de las tareas” (las restantes). La diferencia reside en que las páginas del kernel estarán mapeadas para todas las tareas, y el resto de las páginas no. Esto se puede ver en la figura 4, que muestra el mapeo de páginas para una tarea cualquiera.

Al pedir una página de memoria, la misma se debe marcar en pantalla como utilizada. Para esto, se pintará el fondo de un color a gusto y se identificará con un número entre 1 y 8 correspondiente a la tarea que pidió. Se debe recordar que tanto el kernel como la tarea especial *idle* se identifican con el número 0. El color de fondo que identifique una página pedida, debe ser el mismo para todas las páginas pertenecientes a la misma sección, ya sea de kernel o de tareas.

4.3. *quantum* en el Scheduler

El *Scheduler* a implementar es muy rudimentario. Su único trabajo es el de intercambiar tareas cada una determinada cantidad de ticks. Todas las tareas contarán con un contador de *ticks*; cuando una tarea está corriendo ese contador disminuye por cada *tick* de reloj. Una vez que llega a cero se produce el intercambio por la tarea siguiente. Este proceso se repite hasta que no existan más tareas por correr, momento en que se debe correr una tarea especial denominada *idle*.

Inicialmente todas las tareas tienen su contador de *ticks* seteado en 5. Por medio del teclado se puede incrementar(↑) o decrementar(↓) este valor; el cual varía entre 5 y 95, con pasos de a 5 unidades. Las teclas para controlar el *quantum* de cada proceso son las siguientes:

Acción	`tarea1`	`tarea2`	`tarea3`	`tarea4`	`tarea5`	`tarea6`	`tarea7`	`tarea8`
↑	`q`	`w`	`e`	`r`	`t`	`y`	`u`	`i`
↓	`a`	`s`	`d`	`f`	`g`	`h`	`j`	`k`

La tarea especial *idle*, por su parte, tiene un *quantum* fijo de 1.

Por otro lado, el *scheduler* debe presentar en pantalla un reloj por cada tarea que cambia en cada ciclo de reloj. En la figura 3 se puede ver la posición de cada reloj que itera los caracteres "−", "\", "|" y "/". Por último, si una tarea trata de acceder fuera de su área de memoria, se debe marcar la misma con una "x" en el lugar de su reloj.

5. Ejercicios

5.1. Ejercicio 1

a) Completar la Tabla de Descriptores Globales (`GDT`) con 2 segmentos, uno para código y otro para datos, que ocupen todo el espacio de 4 GB. Ambos deben ser segmentos de nivel 0 y ocupar los índices 6 y 4 respectivamente. Además, se debe completar el índice 2 con un segmento que direccione la memoria de video.

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

c) Escribir una rutina que se encargue de limpiar la pantalla y pintar la misma con colores como indica la figura 5. Para hacer esto se debe direccionar a memoria utilizando el segmento en el índice 2.

> **Pregunta 1:** ¿Qué ocurre si se intenta escribir en la fila 26, columna 1 de la matriz de video, utilizando el usando el segmento de la `GDT` que direcciona a la memoria de video? ¿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`. Trabajar sobre los archivos `gdt.c` y `kernel.asm` .

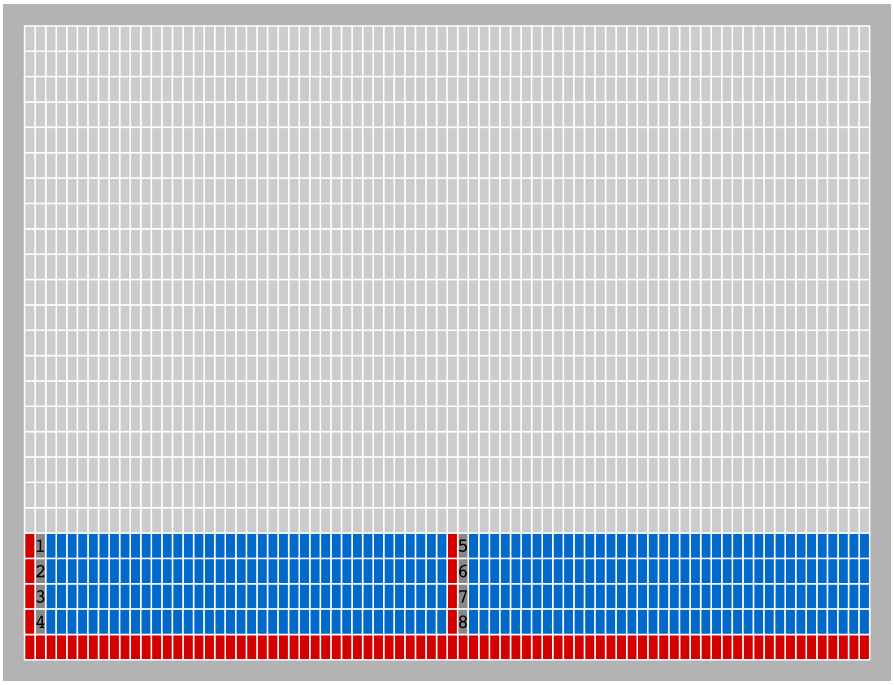


Figura 5: Pantalla en modo protegido

5.2. 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. Para escribir en pantalla, las rutinas deben hacerlo directamente sobre la memoria de video.

b) Escribir una función que imprima en pantalla todos los valores de los registros del CPU y los últimos 6 elementos de la pila. Además debe imprimir en pantalla el *backtrace* de llamados a funciones, es decir, las direcciones de retorno almacenadas en la pila a la hora de armar el *stack frame*. El límite es de 5 llamados. Considerar inicialmente el valor de `ebp` como 0. Esta función se debe aplicar a las rutinas de atención de interrupciones del ejercicio anterior. En la figura 6 puede verse el resultado si se produce una excepción de doble falta.

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

Pregunta 2: ¿Cómo se puede hacer para generar una excepción sin utilizar la instrucción `int`? Mencionar al menos 3 formas posibles.

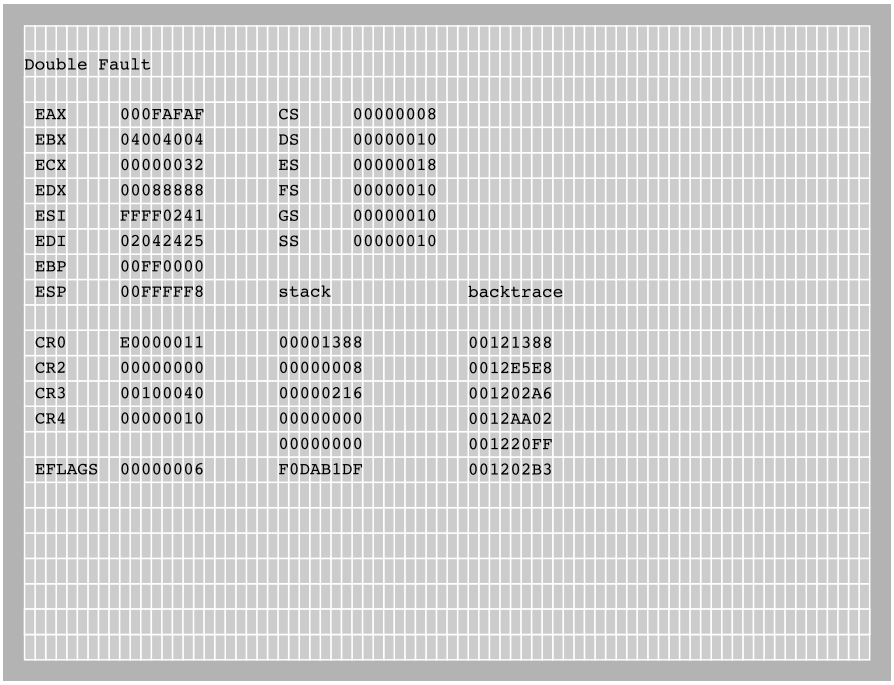


Figura 6: Ejemplo de como imprimir los registros del sistema

Pregunta 3: ¿Cuáles son los valores del stack? ¿Qué significan?

Pregunta 4: ¿Puede ser construido el *backtrace* si no se construye el `stack frame` en cada llamado a función? ¿Por qué?

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`. Trabajar sobre los archivos `idt.c`, `isr.asm` y `kernel.asm` .

5.3. Ejercicio 3

a) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el Kernel (`inicializar_dir_kernel`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x0019FFFF`, como ilustra la figura 1. 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 el centro de la última fila de la pantalla.

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

Pregunta 6: Un espacio de direcciones virtuales de 4Mb se encuentra mapeado utilizando una página de ese mismo tamaño. Sin embargo se desea tomar 4Kb dentro de ese espacio de direcciones y mapearlos en un espacio físico distinto. ¿Es esto posible? ¿Cómo? Realizar pseudocódigo de ser necesario.

Pregunta 7: Suponiendo que una tarea pueda modificar el directorio de páginas, ¿puede esa tarea acceder a cualquier posición de la memoria?

Pregunta 8: ¿Puede una tarea en un sistema operativo completo, escribir todas las direcciones de memoria de los 4Gb que le es posible direccionar? ¿Qué dificultades genera?

Nota: Trabajar sobre los archivos `mmu.c` y `kernel.asm` .

5.4. Ejercicio 4

a) Escribir una rutina (`inicializar_mmu`) que se encargue de inicializar las estructuras del mapa de memoria.

b) Escribir dos rutinas encargadas de pedir páginas de memoria tanto para el espacio de páginas de kernel como de tareas. Las mismas pueden respetar la siguiente aridad respectivamente, `unsigned int malloc_page_K(unsigned int process_id)` y `unsigned int malloc_page_T(unsigned int process_id)`.

c) Escribir dos rutinas encargadas de liberar páginas de memoria previamente pedidas tanto para el espacio de páginas de kernel como de tareas. Las mismas pueden respetar la siguiente aridad respectivamente, `void free_page_K(unsigned int fisica)` y `void free_page_T(unsigned int fisica)`.

d) Escribir una rutina (`inicializar_dir_usuario`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea. Se debe configurar, usando *identity mapping*, las direcciones `0x00000000` a `0x0019FFFF`. Esta estructura utilizará páginas libres del kernel.

e) Escribir dos funciones:

I- `mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica)`
Permite mapear la página física correspondiente a `fisica` en la dirección virtual `virtual` utilizando `cr3`.

II- `unmapear_pagina(unsigned int virtual, unsigned int cr3)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

f) Construir un mapa de memoria utilizando la función del ítem *b)* e intercambiarlo con el del kernel, luego cambiar el color del fondo del primer caracter de la pantalla.

Pregunta 9: ¿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 10: ¿Es posible desde dos directorios de página, referenciar a una misma tabla de páginas?

Pregunta 11: ¿Que es el TLB (Translation Lookaside Buffer) y para qué sirve?

Nota: Los archivos de trabajo son `mmu.c` y `kernel.asm` . Por la construcción del kernel, las direcciones de los registros `cr3` 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.

5.5. 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 una a la interrupción `36`.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada tick llame a la función `proximo_reloj`. La misma se encarga de mostrar, por cada vez que se llama, la animación de un cursor rotando. En la figura 7 se detalla en que posición de la pantalla aparece el reloj. La función `proximo_reloj` está definida en `isr.asm`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquier número, se presente el mismo en la esquina inferior izquierda de la pantalla. Si el número se deja de presionar se debe volver al carácter previamente escrito en la pantalla. El número debe ser escrito en color magenta con fondo cyan. [1]

d) Escribir la rutina asociada a la interrupción `36` para que modifique el valor de `eax` por 42.

Nota: Los archivos de trabajo son `idt.c` y `isr.asm`.

5.6. Ejercicio 6

a) Definir 10 entradas en la `GDT`. Una reservada para la `tarea_inicial`, otra para la tarea `IDLE` y las 8 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 `0x00012000`. Para la dirección de la pila, se debe obtener una página libre del espacio para tareas y luego mapearla con *identity mapping*. Debe compartir el mismo `CR3` que el kernel.

[1] http://wiki.osdev.org/Text_UI

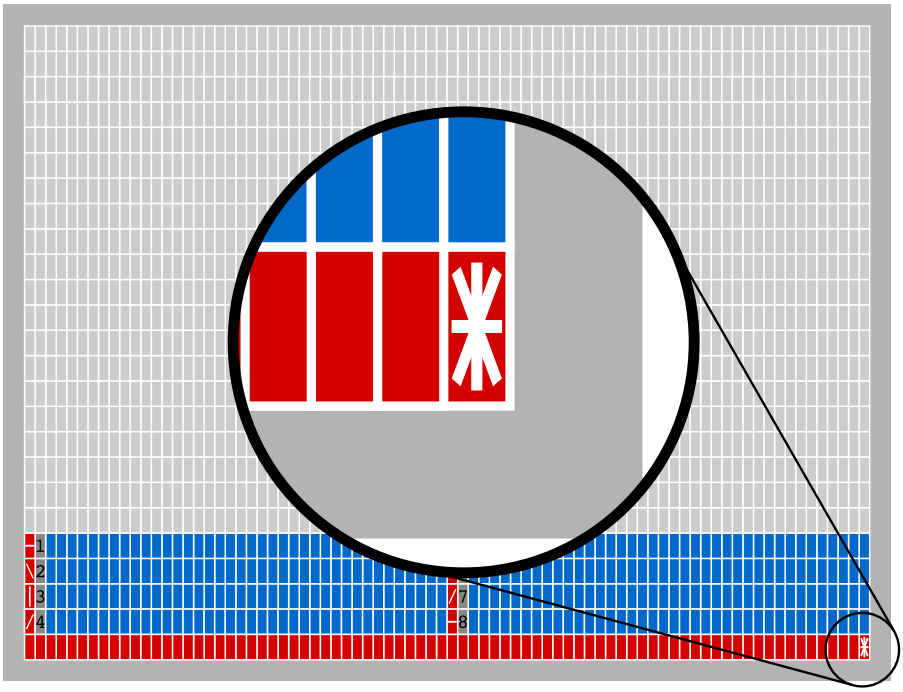


Figura 7: Posición del reloj en la pantalla

c) Completar el resto de las entradas del arreglo de las `TSS` definidas con los valores correspondientes a las tareas que correrá el sistema. El código de las tareas se encuentra a partir de la dirección `0x00013000` ocupando una página de 4kb cada una. El mismo debe ser mapeado a partir de la dirección `0x0`. Para la dirección de la pila, se debe obtener una página libre del espacio de tareas y mapearla a la dirección virtual `0x00030000`. Para el mapa de memoria se debe construir uno nuevo utilizando la función `inicializar_dir_usuario`.

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 8 tareas que se ejecutarán en el sistema.

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 12: 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 13: ¿Cómo se puede verificar si la conmutación de tarea fue exitosa?

Pregunta 14: Se sabe que las tareas llaman a las interrupciones `88` y `99`, por lo que debe realizarse el ejercicio anterior antes de conmutar de tarea. ¿Qué ocurre si no se hace? ¿Por qué?

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

5.7. Ejercicio 7

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

b) Crear la función `proximo_indice()` que devuelve el índice en la `GDT` de la próxima tarea a ser ejecutada.

c) Modificar la rutina de la interrupción `36`, para que implemente los tres servicios del sistema según se indica en la sección 4.

d) Modificar el código necesario para que se realice el intercambio de tareas por cada *quantum* cumplido según cada tarea. El intercambio se realizará según indique la función `proximo_indice()`.

e) Modificar la rutina que atiende el teclado, sin quitar la funcionalidad de imprimir números en pantalla, pero agregando que según la tecla presionada se incremente o disminuya el *quantum* de las tareas. Se debe respetar lo indicado en la tabla de la sección 4. Además, para cualquier cambio en el *quantum* de una tarea, se debe imprimir el nuevo valor en pantalla en algún lugar a elección.

f) Lamentablemente las tareas no están muy bien programadas, y suelen acceder fuera de su espacio de memoria, se debe capturar este error y quitar la tarea del *scheduler*.

Nota: En este caso se trabaja sobre los archivos `sched.c` `isr.asm`y `mmu.c`.

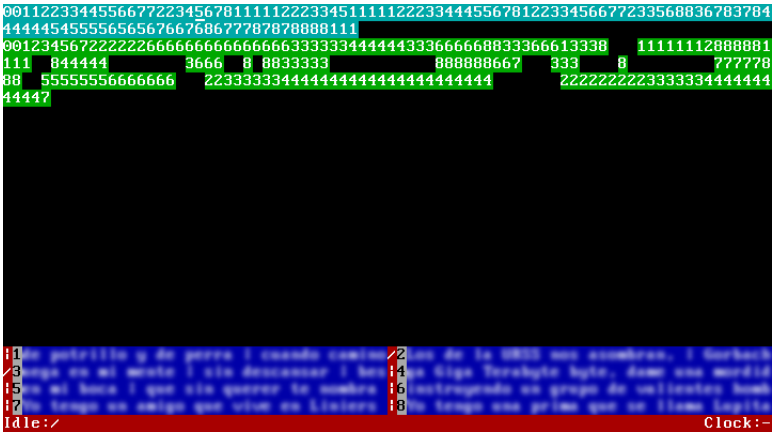
5.8. Ejercicio 8 (optativo)

a) A las tareas les gusta mucho cantar canciones, y qué sería de un tp sin buena musica de fondo. Deben identificar los autores y títulos de cada una de las canciones que las tareas cantan (es decir, imprimen en su buffer).

Nota: Se recomienda no escuchar repetidas veces los temas mencionados a fin de conservar la salud mental.

5.9. Resultado

Una vez funcionando se podrá ver una pantalla como la siguiente,



6. 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 **Martes 19/06** y deberá ser entregado a través de la página web en un solo archivo comprimido. 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.