DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP III - System Programming: TronTank

Grupo: Alemania / Vollkornbrot

Organización del Computador II
Primer Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Quiroz, Nicolás	450/11	nh.quiroz@gmail.com
Rodríguez, Pedro	197/12	pedrorodriguezsjs@hotmail.com
Vuotto, Lucas	385/12	lvuotto@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Resumen

Una computadora, al iniciar, comienza con la ejecución del POST y luego el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En lo que concierne a este trabajo práctico, disponemos de un floppy disk como unidad booteable. En el primer sector de dicha unidad se encuentra el *boot-sector*. El BIOS se encarga de copiar a memoria 512 bytes de este sector, a partir de la dirección `0x7c00`. Luego, se comienza a ejecutar el código a partir de 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 direción `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 ejecutar tareas, es provisto por la cátedra.

Índice

1. Objetivos generales

El objetivo de este trabajo práctico consiste en realizar una serie de ejercicios en los que aplicamos, gradualmente, los conceptos de programación de sistemas, que vimos durante la segunda parte de la cursada. El sistema desarrollado debe ser capaz de capturar cualquier problema generado por las tareas (exactamente 8, a nivel de usuario) y tomar las medidas necesarias para neutralizar las mismas y quitarlas, utilizando los mecanismos propios del procesador, desde el punto de vista del sistema operativo, enfocándonos esencialmente en dos aspectos: el sistema de protección y la ejecución concurrente de tareas.

2. Plataforma de pruebas

Como entorno de pruebas, utilizamos el software ***Bochs***. Este programa, de código abierto, nos permite emular una *IBM-PC*, con arquitectura *Intel x86*, dispositivos comunes de E/S, y un BIOS. También puede ser compilado para emular 386, 486, Pentium/Pentium II/Pentium III/Pentium 4 o una CPU con arquitectura x86-64, incluyendo instrucciones adicionales, como las MMX, SSEx y 3DNow!. Es capaz de ejecutar la mayoría de los sistemas operativos, incluyendo Linux, DOS o Windows NT/2000/XP/Vista/Seven.
Su uso típico es el de proveer una emulación completa de una PC x86, incluyendo al procesador y el resto del hardware y periféricos (memoria, discos duros, teclado, unidad de cdrom, disquetes, etc.), pero también es muy utilizado para la depuración de sistemas (*debugging*), ya que, si el sistema operativo huésped cae, por alguna razón, el sistema operativo anfitrión no cae también. Además, lleva un registro de errores y volcado de archivos. De esta forma, tenemos una *máquina dentro de la máquina*.

Fuente: *Bochs User Manual,* http://bochs.sourceforge.net/doc/docbook/user/index.html

El desarrollo de este trabajo práctico fue realizado, principalmente, en la **máquina 17 del laboratorio 5 del DC**.

- **Sistema Operativo:** Ubuntu Linux 12.04 x86_64, kernel 3.2.0-30-generic
- **Especificaciones del Software:** Bochs IA-32 Emulator Project, versión 2.6.2. [1]
- **Especificaciones del Hardware:** Intel ® Core(TM) 2 Quad CPU Q9650 @3.00 Ghz, 4GB de RAM, memoria caché de 6144 KB.

[1] **Descarga y changelog:** http://sourceforge.net/projects/bochs/files/bochs/2.6.2/

3. Enunciado y solución

Ejercicio 1

a) Completar la Tabla de Descriptores Globales (`GDT`) con 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 733MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 8 posiciones se consideran utilizadas y no deben utilizarse. El primer índice que deben usar para declarar los segmentos, es el 9 (contando desde cero).

En `gdt.c`, creamos un arreglo `gdt`, el cual contiene 13 elementos (en este instante del tp, ya que más adelante agregaremos otras entradas para la TSS y la cantidad total ascenderá a 17). Las entradas de este arreglo son las siguientes:

- `GDT_IDX_NULL_DESC`: Esta es la entrada nula, necesaria para el correcto funcionamiento del mecanismo de la `GDT`.
- Las entradas 1 a 8 están reservadas por restricciones propias del enunciado del tp. El tamaño de las entradas GDT_IDX_CD_0, GDT_IDX_CD_1, GDT_IDX_DD_0, GDT_IDX_DD_1 y es múltiplo de 4KB, valiendo 733MB.

  Para los segmentos mencionado anteriormente, usamos `0x0000` como base y `0xdcff`, es decir, direccionamos 733MB utilizando segmentación *flat*, de manera que los mismos se solapan. Además, seteamos el bit de *granularity* en 1, ya que todos estos segmentos van a ocupar más de 1MB.
- `GDT_IDX_CD_0`: Esta entrada contiene la descripción del segmento de código de nivel 0, que será utilizado para ejecutar el código del *kernel*.
- `GDT_IDX_CD_1`: Esta entrada contiene la descripción del segmento de código de nivel 3.
- `GDT_IDX_DD_0`: Esta entrada contiene la descripción del segmento de datos de nivel 0.
- `GDT_IDX_DD_1`: Esta entrada contiene la descripción del segmento de datos de nivel 3.

b) Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección `0x27000`.

Para entrar en modo protegido, deshabilitamos las interrupciones *(cli)*, habilitamos la *(A20 Gate)*, creamos y cargamos la *GDT* (previamente cargamos en el registro GDTR la dirección base y el límite de la GDT). Luego, activamos el bit correspondiente del `cr0`, y ejecutamos la instrucción `jmp 0b1001000:protected_mode`. Para setear la pila del `kernel`, ejecutamos la instrucción `mov esp, 0x27000`.

c) Declarar un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*.

Agregamos la entrada `GDT_IDX_SD`, que contiene la descripción del segmento de video, del nivel 0, utilizado para la pantalla del juego. Este descriptor de segmento posee *granularity* 0, ya que sólamente ocupa 32KB en memoria. Para que pueda ser utilizado *sólo por el kernel*, seteamos el *descriptor privilege level (dpl)* en 0.

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar en el área de *el_mapa* un fondo de color (sugerido verde). Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior.

- `limpiar_pantalla`: Esta rutina se basa en que el selector `fs` apunta al comienzo del segmento de video `0xb8000`. Recorremos el área de 80x50 por fila y escribimos de a 2 caracteres vacíos (para optimizar la implementación).

- `pintar_pantalla`: Utilizamos un algoritmo muy similar al de
- `pintar_pantalla`, sólo que esta vez, en vez de escribir caracteres, pintamos el fondo (background) de verde.

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.

Completamos las entradas de la `IDT` utilizando un *macro* provisto por la cátedra, con algunas modificaciones. El `dpl` fue seteado en 0. Las entradas están definidas como `Interrupt Gates`, y el selector de segmento es el de código, utilizado por el kernel (entrada 9 de la `GDT`, `GDT_IDX_CD_0`). Una `Interrupt Gate` se utiliza para especificar una rutina de interrupción de servicio, deshabilita el llamado a futuras rutinas de atención de interrupciones, haciéndola especialmente útil, por ejemplo, para atender interrupciones de hardware. Las rutinas de atención de las interrupciones se encuentran en `isr.asm`.

Cada rutina muestra un mensaje alusivo al tipo de excepción generada. Además, de producirse dicha excepción, muestra por pantalla el estado de los registros.

b) Hacer lo necesario para que el procesador utilice la `IDT` creada anteriormente.

Para que el procesador utilice la `IDT`, ejecutamos primero la instrucción `call idt_inicializar`, para inicializar la tabla, y luego `lidt [IDT_DESC]`, para cargarla en el registro correspondiente.

Ejercicio 3

a) Escribir una rutina que se encargue de limpiar el *buffer* de video y pintarlo como indica la figura 8.

Esta rutina simplemente escribe caracteres nulos en el segmento de video.

b) Escribir las rutinas encargadas de inicializar el directorio *page directory* y tablas de páginas *page tables* para el *kernel*
(`mmu_inicializar_dir_kernel`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x00DC3FFF`, como ilustra la figura 5. Además, esta función debe inicializar el directorio de páginas en la dirección `0x27000` y las tablas de páginas según muestra la figura 1.

Para inicializar el directorio de páginas, utilizamos la función `mmu_inicializar_dir_kernel`, la cual, primero crea las 1024 entradas del directorio, con permisos de lectura/escritura y nivel de privilegio 0. Las 4 primeras entradas se marcan como presentes. Además, en estas 4, se cambian las bases, apuntando a `0x28, 0x29, 0x2a, 0x2b`, respectivamente. Luego, inicializamos las tablas correspondientes a cada una de las entradas del directorio, marcando las primeras 3524 como presentes y dejando las 572 restantes como ausentes (no llegamos a utilizar 4 page tables completamente). El directorio es cargado en la dirección `0x27000`, mediante *identity mapping*.

c) Completar el código necesario para activar paginación.

Primero cargamos el directorio de páginas, utilizando las instrucciones

```
mov eax, 0x27000
mov cr3, eax
```

de manera que quede ubicado en la dirección `0x27000`. Luego, activamos el bit de paginación en `cr0`

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

d) Escribir una rutina que imprima el nombre del grupo en pantalla. Debe estar ubicado en la primer línea de la pantalla, alineado a la derecha.

Utilizamos una rutina provista por la cátedra, para imprimir en modo protegido
`imprimir_texto_mp`
que recibe como parámetros: un puntero al string a imprimir, la longitud del string, el color del texto y fondo, y por último las coordenadas fila y columna, respectivamente.

Ejercicio 4

a) Escribir una rutina (`inicializar_mmu`), que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre.

`mmu_inicializar` llama a `mmu_inicializar_dir_kernel` y luego, por motivos organizativos, implementamos un ciclo for, que utilizamos posteriormente para la implementación de los *syscalls*.

b) Escribir una rutina (`mmu_inicializar_dir_tarea`), encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 5 (ver enunciado). La rutina debe copiar el código de la tarea a su área asignada, es decir, sus dos páginas de código dentro de *el_mapa* y mapear dichas páginas a partir de la dirección virtual (`0x08000000`) (128MB).

`mmu_inicializar_dir_tarea` opera de manera similar a `mmu_inicializar_dir_kernel`, pero, a diferencia de esta última, los directorios no comienzan en posiciones arbitrarias, sino que `mmu_inicializar_dir_tarea` obtiene posiciones de memoria libre del *área libre*. Luego, se copia el código de las tareas a alguna posición aleatoria del mapa y mapeamos la dirección `0x8000000` a dicha dirección del mapa. Esta función además devuelve el `cr3` para poder utilizar la estructura de directorios creada.

c) Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.

1. `mmu_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`.
2. `mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`
   Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

- `mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned int attr`
  Mediante el `cr3` recibido por parámetro, accedemos al índice de la tabla del directorio correspondiente en base a la dirección virtual. Si dicha entrada no se encuentra presente, creamos una nueva tabla. Cualquiera sea el caso, al tener la tabla, nuevamente accedemos al índice correcto mediante la dirección virtual. Esta página se marca como presente, se le asignan los atributos de lectura/escritura y usuario/supervisor recibidos como parámetros y le asignamos como base la dirección física que queremos mapear. Finalmente se ejecuta `tlbflush` para invalidar la caché de la `TLB`.
- `mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`: Funciona en manera análoga a `mmu_mapear_pagina`. Una vez determinado el índice de la tabla de directorios en base al `cr3` recibido como parámetro y la dirección virtual, si la entrada **no** está presente, no se hace nada. Sino, se procede acceder al índice correcto de la tabla, de nuevo en base a la dirección virtual, y se marca dicha página como no presente, independientemente de su estado. De nuevo, se ejecuta `tlbflush` con el mismo fin que en `mmu_mapear_pagina`.

Ejercicio 5

a) Completar las entradas necesarias en la `IDT` para asociar una rutina a la interupción del reloj, otra a la interrupción del teclado y por último, una a la interrupción de software `0x52`.

Al igual que en el **ejercicio 2.a**, agregamos 3 nuevas entradas en la `IDT`, asociando las interrupciones de reloj (32) y teclado (33), asignándoles los mismos valores. La interrupción de software (`0x52`), en cambio, se carga igual exceptuando el `dpl`, que esta vez es seteado en nivel 3, de usuario, ya que esta interrupción corresponde a los *syscalls*, que son llamados por las tareas.

b) Escribir la rutina asociada a la interrucpión del reloj, para que por cada *tick* llame a la función `screen_proximo_reloj`. La misma se encarga de mostrar, cada vez que se llame la animación de un cursor rotando en la esquina inferior derecha de la pantalla.

Pudimos manejar interrupciones que no son excepciones y en cada ciclo de reloj llamamos una función que actualiza el clock.

c) Escribir la rutina asociada a la interrucpión de teclado de forma que si se presiona cualquier número, se presente el mismo en la esquina superior derecha de la pantalla. El número debe ser escrito en color blanco con fondo de color aleatorio por cada tecla que sea presionada.

Al igual que en el ítem anterior, escribimos la rutina correspondiente, pero alteramos ligeramente la funcionalidad, de manera tal que, además de mostrar el valor correspondiente en la esquina superior derecha de la pantalla, con el fondo aleatorio, también mostramos el contenido de los registros de la tarea correspondiente.

Ejercicio 6

a) Definir 3 entradas en la GDT para ser usadas como descriptores de TSS. Una será reservada para la tarea_inicial y otras dos para realizar el intercambio entre tareas, denominadas TSS1 y TSS2 respectivamente.

En gdt.c, agregamos 3 entradas para guardar descriptores de la TSS. Como *límite*, pusimos 0x67, *tipo 9* (código), *granularity* en 0, pues utilizamos 104B. Seteamos el *dpl* en 0 y marcamos los segmentos como presentes.

b) Completar la entrada de la TSS1 con la información de la tarea Idle. Esta información se encuentra en el archivo TSS.C. La tarea Idle se encuentra en la dirección 0x00020000. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con *identity mapping*. Esta tarea ocupa 2 páginas de 4KB y debe ser mapeada con *identity mapping*. Además, la misma debe compartir el mismo CR3 que el *kernel*.

Para esto, utilizamos la función tss_inicializar_tarea_idle, que cumple con lo pedido.

c) Completar el resto de la información correspondiente a cada tarea en la estructura auxiliar de contextos. El código de las tareas se encuentra a partir de la dirección 0x00010000, ocupando dos páginas de 4KB cada una. El mismo debe ser mapeado a partir de la dirección 0x08000000. Para la dirección de la pila, se debe utilizar el mismo espacio de la tarea, la misma crecerá desde la base de la tarea. Para el mapa de memoria se debe construir uno nuevo para cada tarea, utilizando la función mmu_inicializar_dir_usuario. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva página libre a tal fin.

Para esto, utilizamos la función tss_inicializar, que se encarga de inicializar los campos de la TSS de cada tarea, segń lo pedido.

d, e y f) Simplemente completamos las entradas correspondientes de la GDT, para la tarea_inicial, TSS1 y TSS2.

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

En kernel.asm, realizamos un jmp far a la entrada de la GDT donde se encuentra ubicado el descriptor de la TSS de la tarea Idle. Para ello, empleamos el siguiente código:

```
offset:   dd 0x0
selector: dw 0x0

...

mov ax, 0b1111000
mov [selector], ax
jmp far [offset]
```

Ejercicio 7

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

En `sched.c`, creamos la función `sched_inicializar`, que se encarga de marcar todas las tareas como vivas e inicializa las siguientes variables de control, con los valores mostrados:

```
_esta_corriendo_la_idle = TRUE;
guardar_tanquecito = TRUE;
_estan_todas_muertas = TRUE;

_tarea_actual = CANT_TANQUES - 1;

primera_vez = TRUE;
```

b) Crear la función `sched_proximo_indice`, que devuelve el índice en la `GDT` de la próxima tarea a ser ejecutada. Construir la rutina de forma que devuelva el índice de la `TSS1` y luego el de la `TSS2` de forma intercalada, para dos tareas fijas.

En `sched.c`, creamos la función `sched_proximo_indice`, que se encuentra generosamente documentada, explicando detalladamente el funcionamiento.

c) Modificar la rutina de la interrupción `0x52`, para que implemente los tres servicios del sistema, según se indica en la sección 3.1.1.

Modificamos la rutina `_isr0x52`, que se encuentra en el archivo `isr.asm`. En esta rutina hicimos un chequeo de los parámetros recibidos, se encarga de brindar los servicios del *kernel* disponibles para las tareas y aplicaciones, desalojando en el caso adecuado.

d) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. El intercambio se realizará según indique la función `sched_proximo_indice`.

Agregamos el código necesario en la interrupción del clock (`_isr32`), para luego llamar a `sched_proximo_indice` y realizar el `jmp far` al descriptor devuelto por dicha función.

e) Modificar la función `sched_proximo_indice` de forma que ejecute todas las tareas según se describe en la sección 3.2.

Lo pedido en este ítem fue realizado conjuntamente en el ítem b, descripto anteriormente.

f) 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.

El *handler* de las excepciones del procesador se encarga de determinar si hubo un cambio de contexto al producirse la excepción, lo cual indica si en ese momento se estaba ejecutando una tarea o no. En el caso que se estuviera ejecutando una tarea, se procede a desalojarla y mostrar por pantalla la razón de desalojo, junto con el contenido de los registros de propósito general, los distintos segmentos y los registros de control. En caso contrario, se imprime por pantalla la excepción producida (en la primer línea) y entramos en un ciclo infinito, "*colgando*" la ejecución del *kernel*.