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). La cantidad total de entradas ascenderá luego a 17. Las entradas de este arreglo son las siguientes:

- `GDT_IDX_NULL_DESC`: Esta es la entrada nula, necesaria para el funcionamiento del mecanismo de la `GDT`.
- Las entradas 1 a 8 están reservadas para uso de Intel. El tamaño de las entradas GDT_IDX_CD_-0, GDT_IDX_CD_1, GDT_IDX_DD_0, GDT_IDX_DD_1 y GDT_IDX_SD es múltiplo de 4KB, valiendo 733MB, salvo la de GDT_IDX_SD, que vale 773MB.
- `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 1.

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, activamos el bit correspondiente del `cr0`, y luego 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.

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 mismas 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.

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 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 `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. El directorio es cargado en la dirección `0x2700`, 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 del mapa, y mapeamos a 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 y teclado, 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.

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.

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.

completar.

d) Escribir la rutina asociada a la interrupción `0x52` para que modifique el valor de `eax` por `0x42`. Posteriormente, este comportamiento va a ser modificado para atender los servicios del sistema.

completar.

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.

completar.

b) .

completar.

c) .

completar.

d) .

completar.

Ejercicio 7

completar.