Organización del Computador II

Segundo Cuatrimestre de 2009

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico 3

Grupo XOR

Integrante	LU	Correo electrónico
Daniel Grosso	694/08	dgrosso@gmail.com
Nicolás Varaschin	187/08	nicovaras22@gmail.com
Mariano De Sousa	389/08	marian_sabianaa@hotmail.com

Índice

1. Introducción

El objetivo del siguiente trabajo es aplicar los conocimientos acerca de programación de sistemas operativos dados en clase. Se escribirá un sistema operativo cuyo propósito sea controlar dos tareas que se ejecutarán a la par. Para ello, el sistema deberá definir todas los segmentos necesarios, usar paginación adecuadamente, escribir en memoria de vídeo, ser capaz de controlar interrupciones y finalmente, ejecutar las dos tareas dadas, asignándole tiempo del procesador a cada una y realizar intercambios entre ellas para crear la sensación de que se ejecutan al mismo tiempo.

Las tareas a ejecutar son el pintor y el traductor. El pintor escribe un mensaje que el traductor leerá y escribirá en pantalla. Cada tarea tiene especificada su dirección de lectura y escritura, la dirección de su pila, a donde escribe su mensaje, etc.

Se utilizará el emulador Bochs para correr y probar el funcionamiento del sistema programado. El emulador posee la capacidad de debuggear convenientemente el código escrito.

2. Desarrollo

El desarrollo del trabajo se realizó en forma gradual, separando en cada ejercicio propuesto: Segmentación inicial, Paginación, Interrupciones y Manejo de Tareas. También se apoyó fuertemente sobre los archivos dados por la cátedra, que definían la estructura general del sistema, las dos tareas ya implementadas, macros y un archivo adicional para habilitar la `Gate A20`, necesaria para el correcto funcionamiento del trabajo.

El archivo principal, que contiene la lógica más relevante del sistema, es el `kernel.asm`. Éste importará y usará los siguientes archivos:

- Segmentación y Paginación
  - `gdt.c`
  - `gdt.h`
  - `kernel-traductor_paging.asm`
  - `pintor_paging.asm`
- Interrupciones
  - `idt.c`
  - `idt.h`
  - `isr.asm`
  - `isr.h`
- Tareas
  - `tss.c`
  - `tss.h`
  - `pintor.tsk`
  - `traductor.tsk`

Se crearon segmentos de código y datos de 4GB de tamaño, y se mapeo la memoria de vídeo. Luego se creó toda la paginación como fue definida en el enunciado del trabajo práctico. El manejo de interrupciones se basa en mostrar un mensaje cuando ocurre una excepción, indicando de que excepción se trata, y detener el proceso del sistema.

Después son agregadas mas entradas en la tabla de descriptores globales (`GDT`) para manejar el switching y ejecución de las tareas dadas, y se definen las `TSS` cuyo propósito es guardar el estado de ejecución de cada tarea.

3. Discusión

3.1. Ejercicio 1

Este primer ejercicio del trabajo práctico consistió en crear una tabla de descriptores globales (`GDT`) básica, para luego pasar el procesador a modo protegido y escribir en la memoria de vídeo el marco sobre el cual estará el resto del trabajo.

El código escrito se dividió en definir la `GDT` en el archivo `gdt.c` y el resto en el archivo principal `kernel.asm`.

3.1.1. `gdt.c`

El archivo consiste en un array de estructuras `gdt_entry` definidas en `gdt.h`. La estructura describe un descriptor global que mide 8 *bytes*. Se compone de bits reservados para el tamaño (`limit`), la dirección base del segmento a describir (`base`), y diversos atributos que serán explicados a continuación junto con los segmentos que definimos:
La `gdt` define cuatro descriptores en el siguiente orden:

- Descriptor nulo
- Descriptor para el segmento de código
- Descriptor para el segmento de datos
- Descriptor para el segmento que apunta a la memoria de video.

Campos de cada descriptor:

- `Limit`: Indica el tamaño de cada segmento definido. En las entradas de código y datos estos *bits* están seteados a su máximo valor, que junto con la base y el *bit* de *granularity* van a formar dos segmentos de 4gb cada uno. En el segmento de video el tamaño es de 4kb, o sea, una página.
- `Base`: Indica en que posición de memoria empiezan los segmentos. Tanto el segmento de código como el de datos tienen como base al `0x0000`, para poder lograr el tamaño final de 4gb. El segmento de vídeo empieza en `0x0B8000` según la especificación dada.
- `Type`: Este campo de 4 *bits*, indica si el segmento es de código o de datos y los permisos de lectura, escritura y ejecución entre otras cosas. Se decidió que el segmento de código lleve el valor `0xA` (según el manual de Intel, esto es segmento de código de lectura/escritura), el segmento de datos `0x2` (segmento de datos de lectura/escritura) y el segmento de video también `0x2`.
- `Bit S`: Indica si el segmento es de código/datos o de sistema. Para todos los segmentos se indica que es un segmento de código/datos.
- `Bit DPL`: Indica el nivel de privilegio que tiene el segmento. Todos los segmentos tienen privilegio a nivel *Kernel* (el máximo posible).

- `Bit P`: Indica si el segmento está presente. Para todos los segmentos, se indica que esta presente, sino al tratar de accederlo se produciría una excepción.
- `Bit AVL`: Irrelevante, seteado a 0.
- `Bit L`: Este *bit* indica si el segmento contiene código de 64 *bits*. Seteado a 0 para indicar que este no es el caso.
- `Bit D/B`: El *bit* funciona de manera diferente según se trate de un segmento de código, datos expand-down, o de stack. Lo único relevante en este punto es que en el segmento de código está seteado en 1 para que interprete direcciones de 32 *bits* y operandos de 32 u 8 *bits*, como es necesario en modo protegido.
- `Bit G`: Este *bit* (`granularity`) indica si el campo `limit` del segmento se interpreta como unidad de *bytes* (dando como máximo 1MB de tamaño de segmento) o como unidades de 4kb (dando el máximo de 4gb buscado). Para todos los segmentos, este *bit* esta seteado.

3.1.2. `kernel.asm`

Luego de habilitar la A20 y detener las interrupciones, se carga el registro `GDTR` que apunta a la tabla de descriptores que fue explicada anteriormente. Para ello se utilizó la instrucción:

```
lgdt [GDT_DESC]
```

A continuación se setea el *bit* 0 del registro de control `cr0`, poniendo el procesador en modo protegido, y se realiza un `jump far` al comienzo del código en modo protegido:

```
mov eax, cr0
or eax, 1
mov cr0, eax
jmp 0x08:modo_protegido
```

El `0x08` le indica al registro de segmento `CS` que apunte a esa dirección desde la dirección que apunta el `GDTR`. Que es donde está el descriptor de segmento de código que definimos antes. Luego se acomodan los demás registros de segmento al segmento de datos y se empieza con el código para escribir en pantalla.

Para escribir en la pantalla se va a usar la instrucción `stosw`. La instrucción escribe un *word* guardado en `ax` en la dirección apuntada por `es:edi`. Entonces se carga el inicio del segmento de vídeo en el registro `es`, después se limpia la pantalla escribiendo `0x0000` en `ax` e incrementando el *offset* de `edi`. El registro `ecx` se utiliza para manejar el *loop*.

```
mov ax, 0x18
mov es, ax
xor edi,edi
mov ecx, (25 * 80)
mov ax, 0x0000
limpiarPantalla:
```

```
stosw
loop limpiarPantalla
```

Posteriormente se realizan las rutinas simples para dibujar el marco de una manera similar a la recientemente mostrada.

3.2. Ejercicio 2

En el ejercicio 2 se pedía crear e inicializar dos directorios y tablas de páginas según los mapas de memoria dados en el enunciado. Los directorios de página fueron definidos como `page_dir_pintor` y `page_dir_kernel` en el archivo `kernel.asm`. Los archivos `kernel-traductor_paging.asm` y `pintor_paging.asm` contienen las tablas ya inicializadas.

Para empezar a usar la paginación, se debe habilitar en el registro `cr0`, pero antes hay que inicializar los directorios y tablas de paginas necesarias.

Para inicializar los directorios se definió la subrutina `page_init` que define la primera entrada de ambos directorios con su primer tabla (`page_table_0` para el directorio del *kernel* y `page_table_0_pintor` para el directorio del pintor). Antes de ser llamada, hay que especificar un valor válido para el esp, que controla el *stack* principal, ya que al hacer el `call` a la función `page_init`, el programa debe conocer la dirección a la cual regresar para continuar la ejecución.

Ya dentro de la subrutina, se les agregan los *flags* de `supervisor`, `read/write` y `present` a las primeras entradas de cada directorio, junto con la dirección correspondiente a su primer tabla.

- `Dirección`: se toman los *bits* 31 al 12 inclusive (20 *bits*) de la dirección de memoria recibida por el módulo de segmentación (dirección lineal) y esto servirá para localizar la tabla de página correspondiente.
- `Bit S`: es el *bit* 7 y determina el tamaño de la pagina, que puede ser de 4Kb o 4Mb, pero para esto último se necesita que el modo `PSE` de paginación extendida esté activado (no requerido en el trabajo actual).
- `Bit A`: `Accessed`, determina si se escribió o leyó de una página o no. Es el *bit* 5.
- `Bit D`: este *bit* deshabilita el caché, por lo que permanecerá en 0. Es el *bit* 4.
- `Bit W`: es para habilitar el modo `Write-through` de la *caché*, no lo utilizamos.
- `Bit U`: determina el nivel de privilegio, si no esta *seteado*, el privilegio es de `Supervisor` (el mas alto).
- `Bit R/W`: habilita la lectura y escritura en las páginas, lo requerimos en el trabajo.
- `Bit P`: dice si la pagina esta presente en la memoria. Lo habilitamos para no producir una `Page Fault`.

3.2.1. Tablas de página

Como se dijo anteriormente, las tablas de página estan definidas en los archivos `kernel-traductor_paging.asm` y `pintor_paging.asm`. Cada tabla de páginas tiene un mapeo de memoria diferente, el cual fue definido a mano usando *macros* para ayudar. En cada tabla se tuvo que hacer *identity mapping* para algunas direcciones, otras fueron mapeadas a lugares de memoria fijos (como en el caso de la memoria de video) y otras no fueron especificadas, esto es, inicializarlas en 0.

Para realizar el mapeo de memoria se utilizaron *macros* que permitieron definir las entradas de la tabla correspondientes fácilmente. A continuación un ejemplo de la macro utilizada para *identity mapping*:

```
%assign i 0x0000
%rep    0x009
    dd  i | 3
%assign i i+4096
%endrep
```

Primero se asigna `i` a la dirección sobre la que se quiere hacer *identity mapping*, después se hace un `or` con el valor 3, que al igual que en los descriptores de directorios, esto quiere decir que la página tendrá privilegios de supervisor, estará presente y será de lectura/escritura, luego se repite n veces con n la cantidad de paginas a *mapear*.

Las direcciones que tenían que ser mapeadas a una página específica fueron definidas individualmente, por ejemplo:

```
%assign i 0x13000
%rep 0x001

dd 0xb8000 | 3

%assign i i+4096
%endrep
```

La diferencia con el código anterior, es que éste tiene especificada la dirección a la cual *mapear*, en vez de usar al variable `i`.

Finalmente las direcciones que no se debían asignar, fueron dejadas en 0. Esto significa que ni siquiera estarán presentes en memoria, ni podrán ser utilizadas. Dejamos, por legibilidad, lineas como `%rep 0x0001` o `dd 0 | 0`. Luego en el *kernel* se completan ambas tablas de página con 0.

Para poder hacer un `call` a `page_init` en el *kernel* se definió antes el valor de la pila de *kernel* en `0x7FFC` (`esp` y `ebp`). Después de tener inicializados los directorios se copia la dirección del directorio de páginas de *kernel* al `cr3` y se habilita la paginación poniendo en 1 el *bit* más significativo del `cr0`.

En la segunda parte del ejercicio, pide escribir el nombre del grupo en la esquina superior izquierda. Para ello, se accede a la memoria de video a través de su respectivo segmento definido en la `GDT`, y se usa un código similar al usado en el ejercicio 1 para limpiar la pantalla, con la diferencia que está la instrucción `lodsb` la cual lee un *byte* de la dirección `ds:esi` y lo guarda en el registro `al`. Previamente el registro `esi` fue apuntado a la dirección a donde se encuentra el

mensaje a escribir. Usando otra vez la instrucción `stosw` se escribe el mensaje en pantalla.

3.3. Ejercicio 3

En este ejercicio, el objetivo era asociar rutinas para controlar todas las excepciones del procesador, definiendo correctamente las entradas de la `IDT`. Luego se debía asociar la interrupción del reloj con una rutina `next_clock` dada por la cátedra. Para lograr esto, se utilizaron varias macros que serán descriptas más adelante. Antes de realizar el manejo de interrupciones, hay que inicializar los `PIC` que mapean cada interrupción recibida con su correspondiente dirección para ser atendida. En modo protegido la configuración inicial de los dos `PIC`, esta *mapeada* a interrupciones reservadas por Intel. Por lo cual, hay que remapear los vectores de interrupción a un espacio no reservado.

Una vez reinicializados los `PIC` con nuevos valores, se procede a llamar a la función `idtFill`, la cual está definida en `idt.c`. Esta función llama sucesivamente a un define (`IDT_ENTRY`) el cual se encarga de inicializar una entrada del array `idt`. Dicha entrada apunta a la dirección definida dentro del archivo `isr.asm`, que imprime un mensaje de excepción, usando la macro `IMPRIMIR_TEXTO` dada por la cátedra.

También se define la `isr32` que tiene el código que se ejecuta cuando se produce una interrupción por el *timer*. Este código llama a la función `next_clock` y salta entre las tareas (esto último será explicado en la siguiente sección).

Por último, volviendo al código principal, luego de la llamada a `idtFill`, se carga la tabla `IDT` con la instrucción `lidt`.

3.4. Ejercicio 4

Como último ejercicio de este trabajo práctico, dado dos archivos binarios de tareas dadas por la materia, fue necesario la implementación del *switcheo* de éstas dos tareas. En pocas palabras, como el procesador no puede realizar varias tareas al mismo tiempo, el sistema intenta simular esta condición, otorgándole tiempo, periódicamente, de uso del procesador (*multitasking*). Se utilizó la interrupción del *tick* del *clock* para realizar este cambio de tareas.

Antes de realizarlo, fue necesario concebir un espacio donde poder guardar el contexto de nuestras futuras tareas, así cuando se estén *switcheando*, al volver a tener el control del `cpu` en un momento dado, puedan continuar desde donde dejaron su trabajo. El archivo `TSS.h` define la estructura del espacio donde guardar el contexto. El archivo `TSS.c` genera un array de tres de estos espacios (llamados `TSS`), el segundo para la tarea pintor, y el tercero para la tarea traductor. La primer `TSS` se utiliza únicamente para habilitar el *multitasking*, que en breve, pasará a explicarse.

Estas `TSS` ubicadas en memoria, necesitan de descriptores particulares (descriptores de tareas) en nuestra `GDT`, para así poder ubicarlos. Se generan en el `gdt.c` tres descriptores de `TSS` entonces, uno para cada una de las recientemente mencionadas. A continuación, un gráfico con las componentes del descriptor.

[MANDAR FOTITO DEL DESCRIPTOR DE TSS]

Al conocer su tamaño (`0x6c` *bytes*), pudo definirse el límite de los descriptores, tanto como sus atributos. Se dejó para el `kernel.asm` el trabajo de setear la base.

Ya ahora en el archivo `kernel.asm` nos encargamos de rellenar la información de las `TSS` del pintor y del traductor. Guardando sus pilas correspondientes (ubicadas en las posiciones dadas por el *mapeo* de memoria de la materia), los *flags* de cada uno (0x0202 para habilitar interrupciones y cumplir con el *bit* número 2 prendido requerido por Intel), guardando en cada `TSS` el correspondiente directorio de páginas de cada uno en el espacio salvaguardado para cada `CR3`; guardando todos los registros de propósito general y los registros de segmentación.

Teniendo las `TSS` con la información necesaria para empezar con el *task switching*, nuestro siguiente paso fue completar los descriptores de tares de la `GDT` con la base de cada `TSS`.

```
mov eax,tsss
add eax,TSS_SIZE ;(TSS de pintor, la siguiente es la del traductor)
mov word [edi+2],ax
shr eax,16
mov byte [edi+4],al
mov byte [edi+7],ah
```

La dirección de memoria del comienzo de la TSS se guarda en cada uno de los descriptores. Las siguientes instrucciones se encargan de cargar nuestro primer descriptor de `TSS` que sólo usamos para iniciar el *multitasking* y así poder saltar a la primera tarea verdaderamente, que en nuestro caso es el pintor. La instrucción `ltr` carga el registro pasado por parámetro en el *task register*, al aplicar entonces el `jump far` hace un *switch* del *task register* a la nueva tarea. Luego la instrucción `sti` habilita las interrupciones.

```
mov ax,0x20
ltr ax
sti
jmp 0x28:0
```

En el archivo `isr.asm` fue modificada la interrupción de clock (`isr32`), para que además de imprimir en pantalla lo necesario para completar el ejercicio 3, también realice el cambio de tareas, es decir, actúe como *scheduler*. Comparando el `CR3` actual de la tarea en curso pudimos constatar si lo que se estaba ejecutando era el pintor o el traductor, haciendo un `jump far` seleccionando el descriptor de la tarea opuesta a la actual, y así generando automáticamente un intercambio de tareas por parte del procesador.

Decidimos utilizar una puerta de interrupción para realizar el *scheduler* porque, al tener que intercambiar únicamente entre dos tareas, era el esquema más sencillo de implementar. La rutina comienza comparando el registro `CR3` con la dirección `0xB000` (directorio de páginas del traductor y *kernel*), para saber qué tarea se está ejecutando actualmente. En el caso de estar en la tarea Pintor la comparación da falsa, y se pasa a ejecutar la tarea Traductor mediante un `jump far` que carga el 5to descriptor de la `GDT`, que le corresponde a la tarea. Si se está ejecutando el Traductor y vuelve a suceder la interrupción entonces la comparación va a resultar verdadera y se realizará el cambio a la tarea Pintor. Algo a tener en cuenta es que al ejecutarse la rutina de atención de la interrupción, el `eip` de la tarea que se estaba ejecutando pasa a apuntar a la rutina de atención. Esto hace que al saltar a la tarea correspondiente, el

`eip` que se guarda en la `TSS` de la tarea que estaba en ejecución se deja en la instrucción siguiente al salto. Cuando se vuelve a ejecutar la tarea, lo primero que hace es habilitar las interrupciones (que habían sido deshabilitadas al principio de la rutina) y luego ejecuta la instrucción `iret` que vuelve el `eip` al lugar donde estaba en el momento que fue interrumpida la tarea.

4. Conclusiones

En el presente trabajo se mostró una implementación simple de un posible sistema operativo, utilizando todo lo aprendido en clase, apoyándose en los conocimientos adquiridos de assembler en la primer parte de la materia. El trabajo realizado da el pie para mejorarlo y crear un sistema funcional más eficiente y útil. Aún así, la complejidad requerida para escribir un sistema operativo es enorme, y muy difícil de corregir en el caso en que aparezca un error, pero le da al programador el poder absoluto sobre el procesador.

La cantidad de estructuras de datos necesarias, y conocimientos específicos y técnicos hacen que programar un sistema operativo no sea algo trivial. Las mayores complicaciones surgen a partir de la compatibilidad que introduce Intel para poder manejar programas de procesadores antiguos. Para terminar, el trabajo expuesto presenta el verdadero funcionamiento y poder del procesador de Intel de 32 bits y los posibles problemas que puede traer el abordar la tarea de diseñar e implementar un sistema operativo.