Organización del computador II
Trabajo Práctico III

Rodrigo Campos (561/06)

July 12, 2011

Detalles de implementación

Ejercicio 1

Para respetar que los segmentos de índice 2 y 3 sean segmentos de código y datos respectivamente se decidio dejar los primeros dos segmentos (índices 0 y 1) nulos. El segmento de índice 4 se utilizó para direccionar a memoria de video, como pide el enunciado. De esta forma, la estructura de la GDT resultante es:

Índice	Segmento
0	NULL
1	NULL
2	Código
3	Datos
4	Video

Para crear la gdt se utilizó el arreglo "gdt" definido en gdt.c. Y para completar cada entrada de la GDT se utilizó la estructura "gdt_entry" provista por la cátedra. Los atributos de cada segmento que vale la pena destacar son los siguientes:

- Indice 02 (segmento de código, de 4gb):
  - Granularidad = 1: toma el valor de limite y lo shiftea 12 lugares, de esta forma se puede tomar un segmento que ocupe todo el espacio direccionable.
  - Type = 0xA: Segmento de código con permisos de ejecución y lectura.
  - Segment Present = 1: El segmento se encuentra presente en memoria.
  - DPL = 0x00: El nivel de privilegio del segmento de codigo corresponde al más bajo numerico y de mayor privilegio.
- Indice 03 (segmento de datos, de 4gb):
  - Granularidad = 1
  - Type = 0x02: Segmento de datos con permisos de lectura/escritura.
  - Segment Present = 1
  - DPL = 0x00
- Indice 04 (segmento de datos, correspondiente al mapeo de video). Como la memoria de video se corresponde con una matriz de 25 filas y 80 columnas, donde cada elemento ocupa dos bytes, el tamaño total es $80 \times 25 \times 2$ bytes. Atributos:
  - Base: 0x0b8000
  - Limite: 80*25*2 - 1

- Granularidad = 0
- Type = 0x02
- Segment present = 1
- DPL = 0x00

Luego de completar la GDT en gdt.c cargamos el puntero a la base de la tabla en el registro GDTR con la instrucción LGDT. Con esto queda cargada la GDT.

El siguiente paso es pasar el procesador a modo protegido. Para hacer se hizo:

1. Poner el 1^{er} bit del registro CRO en 1, que corresponde a PE (Protection Enabled).

2. Cargar en el selector de código CS un segmento de código, ya que hasta el momento se usaba un selector nulo. Para hacer esto debemos hacer un JMP al índice de la GDT correspondiente al segmento. En nuestro caso es 0x10, que es el segmento de índice 2 (luego de dos segmentos nulos).

Después de este último paso el procesador está en Modo Protegido, con lo cual debemos cargar el resto de los registros de selectores. Cargamos en ES el segmento dedicado a escribir en pantalla (0x20), y el resto (DS, GS, FS, SS) con el segmento de datos 0x18.

Para probar que la segmentación funciona correctamente escribimos en pantalla. Ponemos toda la pantalla negra y dejamos dos lineas blancas, la primera y última. Esto implicó hacer 3 ciclos simples que escribieran en la memoria con el selector de video.

Ejercicio 2

Los handlers para las interrupciones/excepciones se crearon en la función "inicializar_idt" usando el macro "IDT_ENTRY" provisto por la cátedra. Para esto fue necesario modificar el macro cambiando el selector que usaba, ya que el segmento de código se encuentra en el índice 0x10 y no 0x8. Este macro crea una entrada en la IDT que ejecutará, para la interrupción número i, la función llamada "_isr$i$". Por este motivo en *isr.asm* hicimos una funcion para cada interrupción con este nombre.

En cada funcion, en una primera instancia, realizamos un *jmp* $ para verificar que esté llegando a este handler, y luego nos enfocamos en el detalle de cada función.

Primero escribimos un mensaje contextual para saber de que interrupcion se habia tratado. Usamos el macro *IMPRIMIR_TEXTO*. Luego imprimimos también el contenido de los registros, usando *DWORD_TO_HEX* para convertir el contenido de los registros a ASCII.

Luego imprimimos el Stacktrace y Backtrace. El Stacktrace fue simple y corresponde al siguiente codigo (tener en cuenta que en 0x1C000 es donde se inicializa el esp/ebp):

```
; fila en pantalla a partir
; de la que empiezo a imprimir
mov ecx, 12
; ebx = cantidad de iteraciones
mov ebx, 0
.print_strace:
  ; fila donde comienza la pila
  ; (si llegue aca no debo seguir)
  cmp esp, 0x1C000
  je .strace_end
  ; vamos a imprimir como
  ; mucho 6 elementos
  cmp ebx, 6
  je .strace_end

  mov eax, [esp + ebx * 4]
  ; convierto a hexa
  ; y copio el registro a una posición de memoria
  DWORD_TO_HEX eax, treg
  IMPRIMIR_TEXTO treg, 8, 0x1A, ecx, 0x16
  inc ebx
  inc ecx
  jmp .print_strace
.strace_end:
```

El caso del Backtrace es un caso similar, ya que debemos solamente imprimir el contenido de las direcciones de retorno. Las direcciones de retorno estan siempre en EBP+4, en la medida que las vamos imprimiendo vamos haciendo POP de los ebp anteriores, hasta llegar a las primeras 6 o al primer ebp.

Siempre al final de cada función que imprime pusimos un *jmp* $ para dejar al procesador "loopeando" con el stack/back trace y el contenido de los registros en pantalla.

Para que el procesador tome estos handlers lo que debemos hacer es cargar el registro del procesador con la instruccion IDTR.

Ejercicio 3

Este ejercicio consistió en activar paginacion con Identity Mapping, es decir, que la dirección virtual X se corresponda con la dirección física X. Antes de activar paginación se crearon los elementos necesarios: un Page Directory y las correspondientes Page Tables. Para realizar los mapeos pedidos se necesitó una sola entrada de Page Directory y una Page Table. Esto se debe a que cada página es de 4K, y para mapear 2MB se necesitan, entonces, 512 páginas.

El PD tiene 1024 entradas de 32bits cada una. Cada entrada contiene un puntero a una PT, aunque en los primeros 12 bits hay algunos flags para atributos. Los atributos pertinentes en este trabajo son los de Present y Read/Write,

y deben estar en 1.

Para cargar el PD pusimos la primera entrada apuntando a la única PT, con los atributos correspondientes. En el resto del PD pusimos entradas nulas.

```
; cantidad de entradas
mov ecx, 1024
mov ebx, PD_ADDR
; flag de entrada de memoria no presente
mov eax, 0x2
.pd:
  ; cargamos todas las PDE como no presentes (nulas)
  mov dword [ebx+ecx*4], eax
  loop .pd
; ponemos los ultimos 2 bits en 1
; (present y read/write)
mov dword [PD_ADDR], PT_ADDR+0x3
```

Luego completamos la PT. El método es similar, la diferencia es que cada entrada es la dirección base de cada página. También tiene los mismos atributos.

Para habilitar la paginación primero se debe completar el CR3 y luego poner en 1 el último bit del CR0. Este último bit es el que hace efectiva la paginación. En CR3 debe existir un puntero a la Page Directory (o sea PD_ADDR).

```
mov eax, PD_ADDR
mov cr3, eax
mov eax, cr0
or eax, 0x80000000
mov cr0, eax
```

Para probar que la paginación realizada funciona correctamente, lo que hacemos es imprimir en la esquina superior de la pantalla el nombre del grupo. La forma de escribir es igual a cuando escribimos luego de habilitar segmentación, pero la diferencia es que esta vez el procesador está usando paginación, y si se mapeo adecuadamente con identity mapping, se escribirá en la pantalla pasando por la unidad de paginación.

Ejercicio 4

Se crearon dos variables globales que apuntan a las siguientes páginas libres de usuario y de kernel. Estas variables son inicializadas por la función *inicializar_mmu* con las primeras páginas de usuario y kernel libres. Luego, en las funciones *pagina_libre_usuario* y *pagina_libre_kernel*, lo que se hace es devolver el puntero correspondiente e incrementarlo para que apunte a la siguiente página (manteniendo el invariante de que apunta a la siguiente página libre).

*inicializar_dir_*usuario es una función que cumple un rol similar a lo hecho en el ejercicio anterior, pero en C. Se piden dos paginas libres de kernel a la funcion *pagina_libre_*kernel para usar como PD y PT. Se apunta la primer entrada del PD a la PT y se pone el resto de las entradas como nulas. Luego,

se realiza en la PT el mapeo a las direcciones físicas pedidas. Todos estos procedimientos usando los atributos de present y read/write en 1. Es importante notar que para realizar los mapeos pedidos alcanza con una sola tabla de páginas.

Para probar esta funcion se reemplaza el CR3 con la dirección de la nueva tabla de directorio creada por *inicializar_dir_usuario*, se escribe en pantalla y luego se vuelve al CR3 original. Este proceso verifica que el procesador puede acceder a ambas tablas.

Para mapear/unmapear una dirección virtual cualquiera, en una física utilizando un PD dado, se crearon las funciones: *mapear_pagina y unmapear_pagina.*

mapear_pagina agrega los registros correspondientes en la PD y PT para mapear una dirección virtual a una física. Realiza el siguiente procedimiento:

- Obtiene el offset correspondiente a la entrada de directorio de la direccion virtual. Son los 10bits más significativos.
- Obtiene el offset para la entrada de tabla. Desde el bit 21 hasta el 12 inclusive.
- Con los dos offset de arriba sabemos en qué posición debemos colocar cada entrada.
- Obtenemos de la PD, la entrada correspondiente, y nos fijamos si ya tiene ingresada una tabla. Sino pedimos memoria para esa tabla y hacemos que esta entrada apunte a esa tabla.
- En la tabla, vamos hasta el offset correspondiente a la tabla de la direccion virtual y allí guardamos el puntero a la memoria que recibimos por parametro. También poniendo los flags correspondientes.

unmapear_pagina es similar pero solo recibe por parámetro la dirección virtual. Básicamente llega a la entrada de la tabla de páginas igual que *mapear_pagina*, solo que en vez de apuntarla a una dirección física pone la entrada toda con ceros. Luego de hacer esto, esa dirección virtual de memoria deja de ser accesible.

Ejercicio 5

Para que las interrupciones por hardware lleguen con un valor que no se pise con las excepciones ya definidas por el procesador fue necesario reconfigurar el PIC. Para esto se usaron las funciones provistas por la cátedra.

Se agregaron también handlers para las interrupciones 32 (clock), 33 (teclado), 66, 88 y 89 en la función *inicializar_idt*. En todos los handlers de interrupciones por hardware se tiene en cuenta que hay que resguardar los registros de propósito general y notificar al PIC que se atendió la interrupción.

En el handler de la interrupción de reloj simplemente se llama a la funcion pedida para que dibuje el clock.

En el handler de la interrupcion de teclado se usó los scancodes de las teclas para reconocer si se presionó un número e ignorar la teclas sino. Para imprimir por pantalla se utilizó, nuevamente, el macro IMPRIMIR_TEXTO. Vale la pena

aclarar que solo se usaron los números del teclado qwerty que se encuentran sobre las letras y no los del teclado numérico.

Ejercicio 6

Se utilizó una variable global, *next_entry*, que apunta a la próxima entrada libre en la GDT. Se creó la función *inicializar_gdt* donde se asigna esta variable a la próxima entrada libre en la gdt (teniendo en cuenta las entradas ya ocupadas con los segmentos de código, datos y video) y la función *entrada_libre_gdt* que devuelve el valor actual de la variable *next_entry* y la incrementa para que apunte a la próxima posición. Un procedimiento muy similar al realizado con la unidad de MMU.

También se creo la función *cargar_tarea_gdt* que recibe un puntero a una estructura tss y crea un selector en la gdt usando los siguientes atributos:

- límite = 0x67
- tipo = 0x9
- presente = 0x01

Y el resto de los atributos (salvando el base_address) en cero.

La función obtener_tss_inicial usa la estructura tarea_inicial que la inicializa toda en cero y pone los eflags en 0x202 (es decir, con las interrupciones habilitadas).

Para completar la tarea IDLE se usó obtener_tss_idle. Realiza lo mismo que obtener_tss_inicial pero con la estructura tarea_idle. También pide una página de usuario que será utulizada para la pila de la tarea. Debido a que la pila "crece hacia arriba", se la apunta a la última dirección accesible de la página alineada a 4 bytes.

Para ejecutar la tarea IDLE primero cargamos el task register con la tarea inicial (porque antes de cambiar a la tarea idle, guardara el contexto actual en el tss apuntada por el registro tr) y luego hacemos un jmp para cambiar a la tarea IDLE.

Ejercicio 7

En la funcion crear proceso se piden dos páginas libres de usuario: una para el código de la tarea a ejecutar y otra para la pila. Como la pila crece hacia arriba se la apunta a la última dirección accesible alineada a 4 bytes. Luego, se copia el código apuntado por el parámetro "cargar_desde" en la página que se pidió para el código (como esa direccion virtual no es accesible con el cr3 en uso, se hace un mapeo temporal para poder acceder). También se crea un page directory para la tarea con identity mapping salvo para las direcciones que el enunciado pide otro mapeo, que se respeta. También se crea una entrada nueva para un tss en memoria, se lo completa con valores coherentes (eflags con interrupciones habilitadas, la pila, el EIP y segmentos) y se la almacena

en la gdt. Así mismo se guarda en el arreglo "tareas" el selector de la gdt para facilitar la implementación de "proximo_indice". La forma de calcular esto puede no parecer inmediata, pero es bastante sencilla, simplemente le suma a las tareas ya creadas con crear_proceso (la variable "idx_tarea_libre") 7 que son las entradas de la gdt ya usadas (por selectores de segmento y los selectores de tss de la tarea idle e inicial) y luego lo multiplica por 8 para que no sea el índice (1 para la segunda entrada) sino el selector (0x8 para la segunda entrada)

Para obtener una nueva entrada en la tss se realizó un procedimiento muy similar a los usados para obtener entradas libres en otras estructuras: una variable global, una función de inicialización que la apunta al primero libre y una función para obtener la próxima entrada libre que devuelve el valor actual de la variable y luego la incrementa.

Para copiar el código simplemente se re-implementó en C la función memcpy, con el nombre memcpy2 que copia byte a byte (usa que el tamaño de un char es un byte).

"proximo_indice" devuelve el valor del selector de tss (0x8, 0x10, etc) de la proxima tarea a ejecutar (usando el arreglo "tareas" que se completó con los datos adecuados cada vez que se creó un proceso). Cuando llega a la ultima tarea, vuelve a la primera. Usa la variable "tarea_actual" para saber cual es la tarea actual que se esta ejecutando y así saber cuál es la siguiente. También se fija si la tarea de control se encuentra dormida (consultando una variable global que se explica en detalle luego) y si es así, la saltea volviendo a la primera.

En el código del handler de la interrupción de reloj se usó una variable selector y offset definidas en memoria que se escriben con el valor devuelto por proximo_indice. Luego se hace un jmp far a este selector cambiando así de tarea.

Para poner una tarea a dormir y despertarla se utilizó una variable global "dormida" y funciones que ponen en 1 (dormida) o 0 (despierta) esta variable. Entonces en la interrupcion de teclado se agregó código para verificar si la tecla que se presiona es la 't'. Cuando es presionada se llama a la función que "despierta" a la tarea control. Cuando ocurre int66 duerme tarea control (cambiando dormida=1) y cambia de tarea llamando a próximo indice como se explicó anteriormente.

Preguntas

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

Ocurre error de General Protection. Tira este error porque estámos tratando de acceder con un segmento fuera de su limite.

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

1. Haciendo una division por 0 (int 0)
2. Escribiendo con el segmento de video en un offset superior al limite (int 13)
3. Escribiendo en el codigo un Opcode invalido (ej: db 'r') (int 6)

¿Cuáles son los valores del stack? ¿Qué significan?

lo que se ha puesto en la pila, los EFLAGS, CS, EIP y, si corresponde (dependeiendo la excepción), un error code.

¿Puede ser construido el backtrace si no se respeta la convención C? ¿Por qué?

Sí, siempre que haya alguna forma de encontrar los codigos de retornos anteriores. En la convención C esto ocurre y por eso se puede hacer. Pero no es la unica convencion que se puede definir con la que se puede crear un backtrace.

¿Es posible mapear una página de 4Kb que se encuentra dentro de otra página de 4Mb que se ya encuentra mapeada? Realice el pseudocodigo de ser necesario

Si, teniendo en cuenta activar el bit de Page Size en la entrada de directorio para las paginas de 4mb y activar el bit PSE en el cr4 para accederlas. De esta forma, para dos dircciones virtuales distintas (dir_virtual y dir_virtual2) se puede mapear a una misma direccion fisica:

map_page_4mb(dir_virtual, cr3, fisica)
map_page_4k(dir_virtual2, cr3, fisica)

Suponiendo que una tarea pueda modificar el directorio de páginas ¿Puede esta tarea acceder a cualquier posición de la memoria?

Sí. Puede apuntar una entrada de directorio al mismo page directory y si la entrada en la tabla de páginas de la dirección virtual tiene otro índice que la entrada en la tabla de directorios, como puede escribir en el directorio de páginas, puede apuntarla a cualquier posición física de la memoria logrando así acceder. Es importante notar que la limitación está sobre la dirección virtual a usar y no sobre la dirección física, ya que puede acceder a cualquiera. Sin embargo, si esta restricción sobre la dirección virtual molesta y puede pedir memoria, esta memoria la puede utilizar para la tabla de páginas de esas entradas. Y si no puede pedir memoria probablemente puede usar la pila como tabla de páginas. Igualmente, como ya se dijo, todos estos "trucos" son necesarios para usar todas las direcciones virtuales y no para usar acceder a cualquier dirección física.

¿Puede una tarea escribir todas las direcciones de memoria de los 4Gb que le es posible direccionar? ¿Qué dificultades genera?

Hay espacio en memoria usado para la GDT y, si esta habilitada paginacion, para el Page Directory y el Page Table. Tambien el kernel esta puesto en memoria. Luego, si escribe en lo 4gb estaria pisando estas estructuras, el kernel y a la tarea misma. Es decir que una vez que piso su Page Directory (por decir un ejemplo) se generara (dependiendo qué parte de este se haya pisado) una excepcion la proxima vez que intente escribir/leer de memoria.

¿Por qué el rango de direcciones 0x00000000 a 0x00100000 llevan permisos de supervisor? ¿Este permiso de supervisor, va en la entrada de directorio o la entrada de la tabla de páginas? ¿Qué valor debe ir en la entrada de la tabla de directorio? ¿Por qué?

Cada entrada valida de la tabla de directorios y de paginas lleva permisos de supervisor. De esta forma no se deja acceder desde nivel de usuario a ninguna de estas páginas.

¿Es posible desde dos directorios de página referenciar a una misma a tabla de páginas?

Sí. Simplemente en el directorio de tabla de paginas ponemos dos entradas que apunten a la misma tabla de pagina.

¿Que es el TLB (Translation Lookaside Buffer) y para que sirve?

Es una memoria caché que tiene partes de la tabla de paginación. El objetivo es evitar ir a memoria al traducir direcciones lógicas a físicas.

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

Corriendo desde la consola de bochs “info tss” muestra la informacion de la tarea actual. Tambien, corriendo “info gdt” deberia marcar la entrada de la GDT correspondiente a la tarea como Busy.

¿Cómo puede hacer para verificar si la conmutación de tarea fue exitosa?

Poniendo un breakpoint antes de hacer el jmp que cambia el contexto. Ignoro el primer breakpoint (cambio de contexto del kernel a la primer tarea) y luego en el proximo breakpoint se puede correr “creg” e “info tss” y ver que efectivamente cambia respecto de la proxima vez que se detiene por este breakpoint.

Se sabe que las tareas llaman a la interrupciones 88 y 89 y por eso debe realizarse el ejercicio anterior antes de conmutar de tarea. ¿Qué ocurre si no se hace? ¿Por qué?

Si no se pone un handler para esas interrupciones, cuando surja esa interrupcion, se genera un #GP. Esto pasa porque no hay una entrada valida en la IDT. Segun el manual de intel (3A 6-51), ocurrio "Accessing a gate that contains a null segment selector" o "The segment selector in a call, interrupt, or trap gate does not point to a code segment".