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

Trabajo Práctico Nro 3
System Pro1gramming - Batalla Bytal

Organización del Computador II

Segundo Cuatrimestre de 2013

Grupo: **Frambuesa a la Crema**

Apellido y Nombre	LU	E-mail
Ignacio, Truffat	387/10	el_truffa@hotmail.com
Lasso, Nicolás	763/10	lasso.nico@gmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com

Índice

1. Introducción

En el siguiente informe se describen los módulos implementados que constituyen el código del Trabajo Práctico Nro 3 *Batalla Naval*. Cada módulo descripto incluye una breve descripción de las decisiones de diseño tomádas por el grupo con respecto al procesador *Intel* y sus reglas de desarrollo y, de ser necesario, una explicacion de la implementación. Esto incluye en el TP: configuración de la GDT, pasaje a modo protegido, configuración de la IDT, paginación, TSS y la organización del scheduller.

2. Desarrollo y Resultados

2.1. Ejercicio 1. GDT

2.1.1. Global Descriptor Table

Como ya sabemos, el procesador inicia en "modo real", el cual direcciona a 1 MB de memoria y no posee niveles de protección ni privilegios.
Por eso necesitamos que el procesador pase a "modo protegido", para direccionar a más memoria y manejar niveles de protección. El kernel se encargará de hacer esto.
Antes de iniciar en modo protegido, es imprescindible tener bien configurado la Tabla de Descriptores Globales, la cual es una tabla que contiene descriptores de segmento, con la finalidad de definir características de varias áreas de la memoria.
En el enunciado se piden 4 segmentos que deben direccionar a 1.75 GB: 2 para código de nivel 0 y 3 respectivamente, y 2 para datos, de nivel 0 y 3 también.
La estructura de un descriptor de segmento es la siguiente:

- L — 64-bit code segment (IA-32e mode only)
- AVL — Available for use by system software
- BASE — Segment base address
- D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
- DPL — Descriptor privilege level
- G — Granularity
- LIMIT — Segment Limit
- P — Segment present
- S — Descriptor type (0 = system; 1 = code or data)
- TYPE — Segment type

Para definir los segmentos que nos requieren, los items importantes son:

- BASE: este parametro indica el comienzo del segmento. En los 4 casos, este fue 0 ya que se pidió una segmentacion flat.
- P: Present, este parametro indica si el segmento está presente en la memoria. El valor en los 4 segmentos es 0x01 ya que efectivamente estaban presentes.
- DPL: Nivel de privilegios del descriptor. Dado que se piden dos segmentos de código y dos de datos nivel 0 y nivel3, este parametro varía según cual de estos queremos implementar. Nivel 0 implica DPL = 00b y nivel 3 implica DPL = 11b.
- G. Granularity. Este flag indica si el tamaño del descriptor es mayor o menor que 1 Mb. Esto sucede dado que solo se poseen 20 bits para indicar el tamaño del segmento. En particular, si G = 1 entonces el valor de los 20 bits será multiplicado por 4 Kb provocando que con solo 20 bits pueda representar 4Gb de memoria. En nuestro caso queremos un tamaño de 1.75Gb entonces necesitamos G = 1.
- Limit: Tamaño del segmento. Va para los 4 segmentos lo mismo.
  Tenemos que direccionar a 1.75 GB, que son 1792 MB, que equivalen a 1835008 KB.
  Como G vale 0x01, las unidades deben representarse de 4 KB, por eso dividimos por 4.
  $\frac{1835008}{4} = 458752$
  Pero como la memoria empieza desde el 0, debe ser un número menos: 458751
  $458751 = 0x6FFF$

- Type: Indica si es un segmento de código o de datos. Para el segmento de código de nivel 0 ponemos el valor de 0x08, indicando que es "Execute only". para el segmento de código de nivel 3 se usa 0x0A, *Read / Execute*. Mientras que para los 2 de datos ponemos el valor de 0x02, indicando que son de Read/Write.

También se define un segmento que reservado para el área de la pantalla en la memoria. Sabemos que empieza en la dirección base 0x000B8000, con un tamaño de 0x0F9F. Dado que se utilizará como un segmemto de datos, su tipo es de Lectura/ Escritura.

2.1.2. Pasaje a modo protegido

En función de pasar a ejecutar en modo protegido el manual de *Intel*[1] explicita una serie de pasos que se deben seguir para complir con esto.

- Habilitar A20. al realizar esto habilitamos el acceso a direcciones superiores a 1 Mb de memoria.
- Una vez que tenemos configurada la gdt, guardamos su ubicacion en una variable gdt_desc. Para que luego la instrucción lgdt pueda cargar la direccion de comienzo de la GDT.
- Seteamos el flag PE del registro CR0, que indica "Protected Envirnoment".
- Por último para pasar a modo protegido hacemos un jmp al comienzo del segmento de código de nivel 0.
- Una vez ahí acomodamos todos los segmentos apuntando a datos de nivel 0 y seteamos la pila del Kernel en 0x27000 según lo indicado por el enunciado.

[1] Ver Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 System Programming Guide

2.2. Ejercicio 2 IDT

2.2.1. Interrupt Descriptor Table

A través de la IDT, definimos donde está el código de las interrupciones que manejaremos.
La estructura de una entrada en la IDT está definida en idt.h y en idt.c son iniciadas todas las entradas. Por medio de una macro se cargan las primeras 20 interrupciones del procesador, que van desde la división por 0 hasta la interrupción SIMD.
Luego son completadas todas las entradas restantes de la tabla con entradas de interrupciones inválidas con el propósito de manejar de alguna forma todas las interrupciones posibles. Algunas de estas son definidas nuevamente:

- Interrupción 0x32: Clock.
- Interrupción 0x33: Teclado.
- Interrupción 0x50: Servicios del sistema (syscalls).
- Interrupción 0x66: Handlers de las banderas.

En isr.asm se encuentra el código donde atendemos estas interrupciones. Por ahora de la 0 a la 19 sólo se imprime el código de error en pantalla.

La estructura de una entrada de la idt, definida en idt.h, es la siguiente:

- offset_0_15: primeros 16 bits del offset al entry point, que atenderá la interrupción
- segsel: selector de segmento de codigo de nivel 0 la gdt
- attr: atributos de la entrada: Present, DPL, D. Esto varían según si la interrupción es de Reloj o Teclado que llevan DPL = 00b o Servicios o Banderas cuyo DPL = 11b.
- offset_16_31: segundos 16 bits del offset al entry point.

Indice	Descripcion	P	DPL	D
0...19	Ins del procesador	1	0	1
32	Clock	1	0	1
33	Teclado	1	0	1
80	Servicios	1	3	1
102	Banderas	1	3	1

2.2.2. Proceso para activar interrupciones

Para poder activar todas estas interrupciones y sus respectivos handlers se siguen los siguientes pasos:

- Mediante el uso de la instrucción LIDT [IDT_DESC], cargamos el principio del array donde tenemos cargados todas las interrupciones
- Por último se deshabilita, se resetea y se vuelve a habilitar el pic que obtiene las interrupciones.

2.3. Ejercicio 3. Paginación

2.3.1. Kernel, Identity Mapping

Debemos mapear con Identity mapping las direcciones 0x00000000 a 0x0077FFFF. Para esto fueron necesarios:

- 1 Tabla de Directorios de páginas que empieza en la direccion 0x27000.
- 2 Entradas de tabla de directorios que abarcan los 1.75 Gb de memoria.
- 2 tablas de páginas. La primer Page table posee sus 1024 entradas completas direccionando desde 0x00000000 hasta 0x003FFFFF y tiene como base la dirección 0x28000 y la segunda de 0x40000000 a 0x0077FFFF con dirección base en 0x30000.

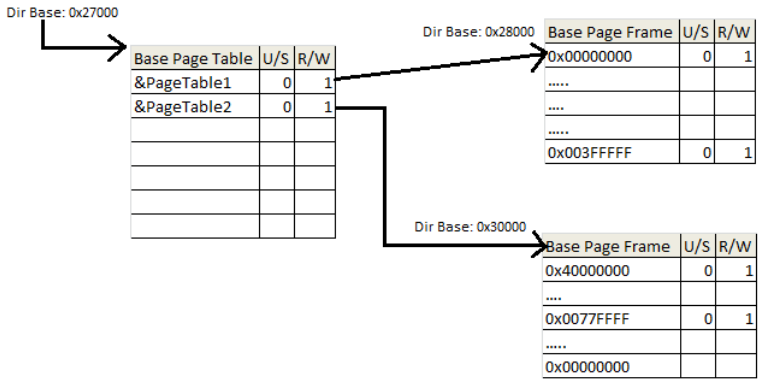
Las entradas de directorio para Kernel son cargadas de la siguiente manera[2]:

- P = 1.
- R/W = 1.
- U/S = 0.
- Direccion de la Page Table = 0x28000 o 0x30000 según corresponda la primer o segunda page table.

Las entradas de Page Table para el Kernel son cargadas de la siguiente manera[2]:

- P = 1.
- R/W = 1.
- U/S = 0.
- Direccion del Page Frame desde 0x00000000 a 0x0077FFFF según corresponda.

A continuación se detalla un esquema para una mejor comprensión de lo explicado:



2.3.2. Activación de paginación

Luego de armar el directorio de páginas podemos habilitar la paginación. Para esto seguimo los siguientes pasos:

- Cargar en CR3 la direccion al inicio del directorio de páginas.
- Setear el bit mas significativo del registro CR0.

[2]Se pueden considerar a los flags no declarados como no seteados, es decir, iguales a 0.

2.4. Ejercicio 4. Paginación de tareas

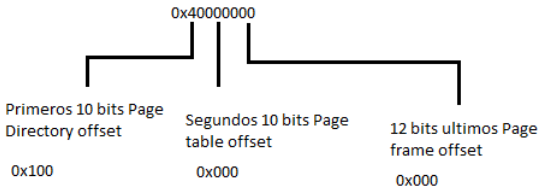
2.4.1. Paginación de las tareas

Para la paginación de tareas se necesitaron los siguientes módulos por cada tarea:

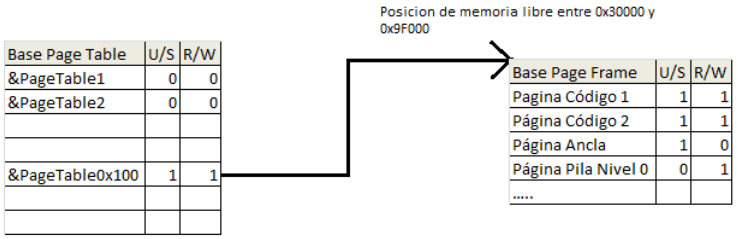
- Inicializar un directorio de páginas con 3 entradas, 2 para el Kernel iguales a las descriptas en el ejercicio 3 y una para direccionar a las páginas de código y pilas de cada tarea. Este Page directory está definido en la direccion 0x40000000
- Dentro de la Page Table de las tareas se encuentran definidas las entradas de cada página de la tarea. Estas son, 2 entradas para el código de la tarea, 1 para el ancla y 1 para la pila nivel 0.

A diferencia del ejercicio anterior, como en este caso estamos mapeando tareas, estas se diferencian de las páginas de Kernel en que los attributos que utilizan son diferentes, es decir, las tareas al correr en nivel 3 requieren que sus Page Directory entries y sus Page table entries posean U/S = 1. De lo contrario no puedo acceder a esas páginas como nos lo demostroó nustra experiencia en la implementación.

El siguiente esquema explica mas simplemente lo nombrado anteriormente. Tener en cuenta que esto debe realizarse por cada tarea y que si bien cada una está mapeada al mismo lugar, la base del Page Directory es distinto para cada tarea, provocando que cada una tenga su propio mapeo. Para empezar, page directory entry que corresponde a la tarea, es la 0x100 ya que 0x40000000 es una direccion virtual entonces la tenemos que decodificar como



Y Luego de esto el esquema de paginación nos quedaría algo así:



Base Page Table	U/S	R/W
&PageTable1	0	0
&PageTable2	0	0
&PageTable0x100	1	1

Base Page Frame	U/S	R/W
Pagina Código 1	1	1
Página Código 2	1	1
Página Ancla	1	0
Página Pila Nivel 0	0	1
.....		

Cabe destacar que para poder cumplir con los ejercicios siguientes, una tarea IDLE va a tener que ser mapeada al comienzo de la paginación de lo contrario, no tener una tarea mapeada me imposibilita arrancar el sistema de scheduling que será introducido mas adelante.[3]

[3] Ver sección Ejercicio 7, Scheduling.

2.5. Ejercicio 5 IDT / Clocks, Teclados, Syscalls y Banderas

La siguiente sección está dedicada a los *Handlers* o manejadores de las interrupciones. Estos son los códigos que se ejecutan cuando alguna porción del systema produce un error o llama a una interrupción mejor conocida como syscall o servicios del sistema.
Nuestro Kernel cuenta con 4 interrupciones que poseen handlers. Estas fueron mencionadas en el punto 3. Procederemos a explicar cada una de ellas:

- Clock: El clock es una interrupción que se ejecuta cada ciertos ticks de reloj. La misma se encarga de buscar el siguiente selector de segmento según es especificado en la sección 7. y realizar el salto a dicho selector que puede corresponder a una tarea, una bandera o la tarea IDLE.
- Teclado: La interrupción de teclado cumple la función de cambiar el estado de la pantalla entre el mapa si la "m" es presionada y el estado de las banderas y las tareas corriendo si la "e" es presionada.
- Servicios o Syscalls: Esta interrupción brinda al systema una serie de servicios o funciones a las tareas:
  - Fondear: Se accede pasando por parámetro el número 0x923.Esta función permite a la tarea mover por tierra el ancla permitiendole mirar de a una página por vez en tierra. Esta llama a la función implementada en C anclar() ubicada en mmu.c.
  - Cañonear: Se accede pasando por parámetro el número 0x83A, la direccion virtual donde voy a disparar y el buffer de 97 bytes que funciona como misil. Básicamente este servicio nos permite escribir en cualquier lugar del mar un buffer de 97 bytes haciendo que en caso de que en esa dirección se encontrara una tarea enemiga, sus páginas sean corrompidas. Esta llama a la función canionear() implementada en mmu.c
  - Navegar: bajo el número 0xAEF, recibe las nuevas direcciones de las primer y segunda páginas de código de la tarea. Generando que mi tarea se pueda mover por el mar sin ser atrapada por una tarea enemiga. Esta syscall llama a navegar() implementada en C en mmu.c.

  En este caso hemos añadido un handler de error extra que verifica que no sean llamadas por una bandera haciendo nuestro código mas seguro.
- Bandera: Esta interrupción se encarga de imprimir la bandera y dar la impresion de movimiento como si una bandera flameara. Cabe destacar que solo puede ser llamada por uan bandera. Si llega a ser llamada por una tarea la misma debe ser desalojada. Esta llama a Bandera() implementada en sched.c

Cabe destacar que las funciones implementadas en C para las syscalls, navegar, canionear y anclar, se encuentran allí dado que manejan páginas de memoria haciendo que ubicarlas en mmu.c sea lo más conveniente para aprovechar todas las funciones y estructuras utilizadas. En el caso de Bandera() se encuentra en sched.c por un razón similar.

2.6. Ejercicio 6. TSS

Para que el procesador pueda despachar, ejecutar o suspender una tarea, es necesario salvar el estado de la misma. La arquitectura provee mecanismos para esto. El segmento de estado (TSS, Task State Segment), es el que se encarga de almacenar la información del estado de una tarea.
Una tarea está identificada por el selector de segmento de su TSS. Y a su vez la TSS es un segmento, por lo tanto debe estar descripto en la GDT junto con los descriptores de segmento de código y datos.
Tenemos 8 tareas y definimos un total de 18 TSS, uno para cada tarea, uno para cada bandera de tarea, uno para la tarea Idle, y otro lo dejamos en blanco para la tarea inicial donde se hace el primer salto.
Las entradas de tss idle y la que está en blanco tienen privilegio de kernel, mientras que las demás están configuradas con privilegios de usuario.
La tarea y la de su flag tienen TSS muy similares, exceptuando por cosas como el eip o donde empiezan las pilas, pero en el resto son iguales ya que comparten casi todo.

2.7. Ejercicio 7. Scheduller