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, Nosecuanto	874/10	el_truffa@hotmail.com
Lasso, Nicolás	763/10	lasso.nico@gmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com

Índice

1. Introducción

2. Desarrollo y Resultados

2.1. Ejercicio 1. GDT

2.1.1. Global Descriptor Table

Como ya sabemos, el procesador comienza en lo que se llama "modo real", que direcciona a 1 MB de memoria y no existen 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. Ubicación del byte 0 del segmento en el espacio de direcciones lineales. El valor en los 4 segmentos es 0x00000000.
P. Present. Indica si el segmento está presente en la memoria. El valor en los 4 segmentos es 0x01.
DPL. Descriptor de privilegios. Según el enunciado, 2 segmentos llevan el valor 0x00, y los otros 2 0x03.
G. Granularity. Debe estar en 0x01 para que las unidades de base y limit se interpreten de 4-KBytes.
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
$458751 = 0x6FFFF$ Type. Indica si es un segmento de código o de datos. Para los 2 de código ponemos el valor de 0x08, indicando que son ."Execute only". Mientras que para los 2 de datos ponemos el valor de 0x02, indicando que son de Read/Write.

También debemos colocar un segmento que describa el área de la pantalla en la memoria. Sabemos que empieza en la dirección 0x000B8000, con un tamaño de 0x0F9F.

Una vez que tenemos configurada la gdt, guardamos su ubicacion en una variable gdt_desc. Para que luego la instrucción lgdt pueda cargar la gdt.
Ya podemos pasar a modo protegido, poniendo en 1 el bit menos significativo del registro CR0, que indica "Protected Envirnoment".

2.2. Ejercicio 2. IDT

2.3. Ejercicio 3. Paginación

3. Conclusión

Las instrucciones SIMD (Single Instruction Multiple Data) proveen al programador de una herramienta más efectiva para realizar el mismo conjunto de operaciones a una gran cantidad de datos.

La aplicación de filtros a imágenes era un ejemplo perfecto para probar su eficiencia.

Analizando los resultados de las implementaciones de los 3 filtros, podemos notar:

- Las operaciones básicas (padd, psub, pmul, pdiv, shifts, etc.) SIMD tienen un costo similar a sus correspondientes operaciones unitarias, pero generalmente requieren algún tipo de pre-proceso para poder trabajar con los 16 bytes (pack, unpack, shifts) en una sola iteración, por lo tanto, aunque más eficientes, no lo son en una relación directamente proporcional.
- En el caso que sí hay una relación directamente proporcional es en el acceso a memoria.
- Además, el acceso a memoria es, por lejos, la operación más costosa de las que implementamos en cada filtro.
- Por consecuencia directa del ítem anterior, las llamadas a otras funciones (que a su vez, probablemente contengan variable locales) dentro de una iteración provocan estragos en la efectividad de las implementaciones en C.
- Para poder aprovechar las instrucciones SIMD es un prerequisito que los datos estén contiguos en memoria.

Concluimos que, definitivamente, las instrucciones SIMD, cuando pueden aprovecharse, demuestran una gran eficiencia. Sin embargo, hay que tener algunas consideraciones:

Aunque las imágenes, video y sonido son los primeros candidatos a ser optimizados por paralelización, no todos los procesos pueden ser efectivos y se requiere un análisis profundo de los datos para ver si vale el esfuerzo.

Además, aunque se pueda lograr una gran optimización, no siempre es lo más importante. La optimización seguramente es indispensable en transmisiones de video en vivo, pero baja en importancia si tuviese que ser aplicado una sola vez en una aplicación tipo MS Paint.

Las desventajas que podrían opacar a la optimización son:

El código no es portable, únicamente funciona en procesadores que implementan el set de instrucciones AMD64, requiriendo reescrituras para otras plataformas. Sin embargo el código C debería funcionar perfectamente en IA-32, ARM y cualquier otro procesador que tenga un compilador de lenguaje C.

El código es mucho más largo y difícil de entender (por lo tanto mayor posibilidad de tener bugs) que en un lenguaje de más alto nivel como C. Y en pos de la optimización, se llegan a eliminar funciones (poniéndolas inline), lo que genera código repetido, largo y confuso.