Organización del Computador II

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

Trabajo Práctico 2: System Programming

Primer Cuatrimestre de 2009

Grupo "UNPCKHPD"

Pablo Antonio	290/08	pabloa@gmail.com
Pablo Herrero	332/07	pablodherrero@gmail.com
Estefanía Porta	451/04	estef.barbara@gmail.com

Índice

1. Introducción

Los procesadores de la *arquitectura IA-32* proveen soporte, estructuras y mecanismos, para la implementación de:

- ciertas técnicas para el manejo de memoria,
- un sistema de administración de tareas,
- la atención de interrupciones y excepciones,
- entre otras.

Los sistemas operativos que se escriban para esta arquitectura pueden, entonces, sacar provecho del soporte que provee el procesador. Es el caso del sistema operativo *JOS*.

El presente trabajo práctico se basa en *completar ciertas funciones faltantes* en una versión del sistema operativo JOS provista por la cátedra. Dichas funciones se orientan a:

- generar un *stack backtrace*, el cual podrá ser invocado mediante un comando en el *kernel monitor* (la parte interactiva del kernel que provee una línea de comandos)
- proveer cierta funcionalidad necesaria para *administrar el sistema de paginación* que utiliza el sistema operativo

1.1. *Stack backtrace*

Examinemos un fragmento de código en C, y cuál sería un posible *assembly* de x86 resultante luego de su compilación:

```
void a() {
    int x = 0;
    b(1);
}

void b(int y) {
    int z = y;
    // breakpoint
}
```

```
a:
    push ebp
    mov ebp, esp
    sub esp, 4
    mov [ebp-4], 0
    push 1
    call b
    add esp, 8
    pop ebp
    ret

b:
    push ebp
    mov ebp, esp
    sub esp, 4
    mov ecx, [ebp+8]
    mov [ebp-4], ecx
    ; breakpoint
    add esp, 4
    pop ebp
    ret
```

Si hiciéramos una llamada a la función `a()`, y nos detuviéramos a observar el estado del *stack* en ese instante, encontraríamos lo siguiente:

	...	
	0	← variables locales de la función
$D_1 \rightarrow$	1	← parámetros para la función a llamar
	`eip` (`ret` de `a`)	← dirección de retorno a la función
	`ebp` de a	← `ebp` de la función llamada
	$[D_1] == 1$	← variables locales de la función llamada
	...	

Es decir, si se respeta la convención de llamadas del lenguaje C, puede interpretarse al stack como una secuencia de *stack frames* de las funciones, conformados, cada uno de ellos, por:

- el `ebp` correspondiente a la función
- las variables locales de la función (si las hay)
- los parámetros para la función que se llamará (si se requieren)
- la dirección de retorno, que será el valor de `eip` una vez que retorne la función llamada (si es que se llama a una función)

Se pedía escribir la implementación de una de las funciones faltantes, que se puede invocar desde el *kernel monitor*, llamada `mon_backtrace()`. Esta función se encarga de realizar un *stack backtrace*, es decir, mostrar la secuencia...

1.2. Administración de memoria: Segmentación

La *administración de memoria mediante segmentación* se implementó desde hace mucho tiempo atrás en la familia de procesadores de Intel. En los 8086 de Intel (que vieron el mercado en 1978), la segmentación, no obstante, fue implementada de una manera rudimentaria, únicamente con los fines de acceder a posiciones de memoria cuya dirección no cabía en los registros del procesador. Por esos tiempos, no se utilizaba la segmentación como un mecanismo para proteger espacios de memoria.

Con la aparición del procesador 80286 (1982) – inclusión del *modo protegido* – se hizo posible la definición de los tamaños y privilegios asignados a cada uno de los segmentos. Así, la segmentación pudo utilizarse para realizar protección de memoria, es decir, restringir el acceso de determinadas tareas a ciertos sectores de la memoria, logrando así, por ejemplo, que las aplicaciones no puedan acceder o modificar datos pertenecientes al sistema operativo.

Sin embargo, en una gran cantidad de sistemas operativos modernos (también es el caso de JOS), la segmentación fue dejada de lado como mecanismo de protección de memoria. Se utiliza, para esto, únicamente la *paginación*.

No obstante, la segmentación no puede ser desactivada en la arquitectura IA-32, en ninguno de sus modos. ¿Cómo hacen entonces los sistemas operativos modernos (que no precisan de la segmentación) para desactivarla? En realidad, no lo hacen. Utilizan algo llamado "Modelo flat de segmentación", que consiste en ubicar a todos los segmentos ocupando todo el espacio que se pretende direccionar, desde la dirección cero. De este modo, la dirección (la parte del offset específicamente) virtual (la utilizada en el código) coincide con la dirección lineal (la que toma como entrada el módulo de paginación).

1.3. Administración de memoria: Paginación

La verdadera protección de memoria en JOS (y en muchos sistemas operativos modernos) se da gracias al *mecanismo de paginación*. La paginación se caracteriza por organizar la memoria física en bloques de tamaño fijo, no solapados, llamados *marcos de página*.

Observemos las distintas etapas por las cuales pasa una *dirección virtual* hasta que se convierte en una *dirección física*:

dirección virtual	$\Longrightarrow_1$	dirección lineal	$\Longrightarrow_2$	dirección física

La primer traducción (1) la realiza la *unidad de segmentación*, mientras que la segunda traducción (2) es realizada por la *unidad de paginación*.

En la arquitectura IA-32 la paginación puede activarse[1] una vez hecho el cambio a modo protegido. Cuando se encuentra activada, y se utilizan páginas[2] de 4KB, la dirección lineal es dividida en tres partes por la unidad de paginación:

índice en el directorio de páginas	índice en la tabla de páginas	*offset*
(10 bits)	(10 bits)	(12 bits)

El primero de los tres campos representa un índice en el *directorio de páginas*. El directorio de páginas es una tabla que contiene 2^{10} entradas (una por cada índice posible). Cada entrada, además de varios atributos, contiene la dirección física de una *tabla de páginas*. El sistema de paginación utiliza el primer campo para seleccionar una de las entradas en el directorio de páginas. Consecuentemente, se obtendrá la dirección física de la tabla de páginas asociada a dicha entrada.

El segundo campo es utilizado, entonces, para elegir una de las 2^{10} entradas de la tabla de páginas mencionada. Las entradas en la tabla de páginas contienen, además de varios atributos, la dirección física de una página en memoria. El *offset* es utilizado para seleccionar uno de los bytes en dicha página.

[1] escribiendo el valor 1 en el bit 31 del registro de control `CR0`

[2] En el código fuente del sistema operativo JOS, se llama páginas a los marcos de página.

2. Desarrollo

3. Conclusiones

Referencias

- Intel 64 and IA-32 Architectures Software 1: Basic Architecture
- Intel 64 and IA-32 Architectures Software 2A: Instruction Set Reference, A-M
- Intel 64 and IA-32 Architectures Software 2B: Instruction Set Reference, N-Z
- Documentación del NASM: http://www.nasm.us/doc/
- http://en.wikipedia.org/wiki/Instruction_prefetch
- Intel 64 and IA-32 Architectures Optimization Reference Manual