

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	pabloherrero@gmail.com
Estefanía Porta	451/04	estef.barbara@gmail.com

Índice

1. Introducción	1
1.1. <i>Stack backtrace</i>	1
1.2. Administración de memoria: Segmentación	2
1.3. Administración de memoria: Paginación	3
2. Desarrollo	4
2.1. Estructuras importantes	4
2.2. Funciones desarrolladas	4
2.2.1. Función <code>page_alloc</code>	4
2.2.2. Función <code>page_free</code>	4
2.2.3. Función <code>pgdir_walk</code>	5
3. Conclusiones	6

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*¹ 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
```

¹(en el dialecto de Intel)

Si hiciéramos una llamada a la función `a()`, y nos detuviéramos a observar el estado del *stack* inmediatamente después de ejecutar la instrucción anterior al *breakpoint*, encontraríamos lo siguiente²:

	...	
	0	} variables locales de la función
$D_1 \rightarrow$	1	} parámetros para la función a llamar
	<code>eip (ret de a)</code>	} dirección de retorno a la función
<code>ebp</code> \rightarrow	<code>ebp de a</code>	} <code>ebp</code> de la función llamada
<code>esp</code> \rightarrow	<code>[D₁] == 1</code>	} 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 de los *stack frames* de una manera específica. Para cada función, deben mostrarse:

1. el `ebp` correspondiente a la función
2. la dirección a la cual retornará la función
3. cinco “parámetros” que recibió la función al ser llamada³

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

²En la figura, las direcciones menores son las que se encuentran ubicadas más abajo.

³En realidad, se imprimen cinco valores que son los primeros cinco parámetros de la función *si esta recibe cinco o más parámetros*. Si la función recibe menos de cinco parámetros, algunos de los valores no serán útiles (no son parámetros de la función).

Podría observarse, si existiera, la información de *debugging* correspondiente a la función para conocer el número de parámetros que esta recibe, y así poder mostrar exactamente (ni más ni menos) todos los parámetros de la función. Sin embargo, esto no se pide.

dirección no cabía en los registros del procesador. Por esos tiempos, en la arquitectura de Intel, 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 la desactivan. 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*:

$$\boxed{\text{dirección virtual}} \Rightarrow_1 \boxed{\text{dirección lineal}} \Rightarrow_2 \boxed{\text{dirección física}}$$

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 activarse⁴ una vez hecho el cambio a modo protegido. Cuando se encuentra activada, y se utilizan páginas⁵ 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	<i>offset</i>
(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 2¹⁰ 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

⁴Esto se hace escribiendo el valor 1 en el bit 31 del registro de control *CRO*.

⁵En el código fuente del sistema operativo JOS, y también aquí, se llama páginas a los marcos de página.

entradas en el directorio de páginas (*PDEs*⁶). 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 (*PTEs*⁷) 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.

2. Desarrollo

2.1. Estructuras importantes

El sistema operativo JOS asigna a cada marco de página de 4KB en memoria física, una instancia de la estructura `Page`.

2.2. Funciones desarrolladas

2.2.1. Función `page_alloc`

```
int page_alloc(struct Page **pp_store);
```

Parámetros:

- `pp_store` es un puntero a un puntero a una página. En el puntero que este referencia se alojará la dirección en memoria de la página obtenida.

Archivo en el que se halla la función: `kern/pmap.c`

Descripción: Si quedan páginas físicas disponibles (caso contrario, devuelve un código de error), `page_alloc` obtiene la primera página de la lista `page_free_list` (la lista que contiene las páginas libres), haciendo uso de la conveniente macro `LIST_FIRST`, y luego la elimina de dicha lista mediante `LIST_REMOVE`. Finalmente, la dirección de memoria de la página obtenida se guarda en el puntero refenciado por el parámetro `pp_store`.

2.2.2. Función `page_free`

```
void page_free(struct Page *pp)
```

Parámetros:

- `pp` es un puntero a la página que se quiere volver a insertar en la lista de páginas disponibles.

⁶ *Page Directory Entry*

⁷ *Page Table Entry*

Archivo en el que se halla la función: kern/pmap.c

Descripción: Esta función vuelve a ubicar la pagina apuntada por `pp` en la lista `page_free_list`, simplemente llamando a la macro `LIST_INSERT_HEAD`, que ubica la página al principio de la lista.

2.2.3. Función `pgdir_walk`

```
int pgdir_walk(pde_t *pgdir, const void *va, int create, pte_t
**pte_store)
```

Parámetros:

- `pgdir` es un puntero a un directorio de páginas (en realidad, a su primera entrada), el cual se utilizará para hallar la dirección física de la tabla de páginas asociada a la dirección `va`.
- `va` es la dirección virtual/lineal cuya PTE asociada (en realidad, la dirección de esta) se quiere obtener.
- `create` indica si debe crearse la página asociada a la dirección en caso de que esta no esté presente.
- `pte_store` es un puntero a un puntero a una PTE. En el puntero que este referencia se alojará la dirección de la PTE asociada a la dirección.

Archivo en el que se halla la función: kern/pmap.c

Descripción: Como se usa un *modelo flat para la segmentación*, la dirección lineal de `va` es idéntica a la virtual. De este modo, pueden obtenerse los índices sobre el directorio y la tabla de páginas utilizando directamente las macros `PDX` y `PTX` sobre `va`.⁸

El primer paso es obtener la entrada del directorio (PDE) correspondiente a la tabla de páginas asociada a la dirección. Esta se obtiene fácilmente direccionando sobre `pgdir` y utilizando `PDX(va)` como índice.

sdsdsd ME QUEDÉ ACÁ

Luego si la tabla de paginas estaba presente, simplemente direccionamos sobre la misma utilizando, `PTX(va)`, y guardamos un puntero a la entry obtenida en `pte_store`.

Si la tabla no esta presente, devolvemos un error; a menos q el parametro 'create' sea igual a 1. En tal caso procedemos a reservar memoria fisica para la misma.

Como primer paso obtenemos una pagina libre llamando a la funcion `page_alloc`, si esta no posee mas memoria disponible, retornamos error. Si la llamada tiene exito, obtenemos su direccion fisica mediante, `page2pa`. Y escribimos en la entry del directorio

⁸Estas macros obtienen el índice en el directorio de páginas y en la tabla de páginas respectivamente, a partir de la dirección que se les pasa.

obtenida anteriormente; cual es la direccion fisica correspondiente, y que los campos presente (P) y de escritura (W) estan activos. Finalmente usando esta direccion fisica, calculamos su direccion virtual mediante KADDR, y le sumamos el valor de PTX(va), para obtener la direccion virtual del entry en la pagina de tablas. Valor q luego se guarda en pte_store. Por ultimo seteamos en 0 todos los bytes de la tabla, mediante memset, indicando que ninguna de las paginas de la misma se encuentra presente en memoria fisica.

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