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 **1**
1.1. *Stack backtrace* 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 mon_backtrace 4
2.2.2. Función `page_alloc` 5
2.2.3. Función `page_free` 5
2.2.4. Función `pgdir_walk` 5
2.2.5. Función `page_insert` 6
2.2.6. Función `page_lookup` 7
2.2.7. Función `page_remove` 8

3. Conclusiones **8**

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 siguiente[2]:

	...	
	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` $\rightarrow$	`ebp` de a	} `ebp` de la función llamada
`esp` $\rightarrow$	$[D_1] == 1$	} 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 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 llamada[3]

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

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

[3] En 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*:

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[4] una vez hecho el cambio a modo protegido. Cuando se encuentra activada, y se utilizan páginas[5] 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

[4] Esto se hace escribiendo el valor 1 en el bit 31 del registro de control `CR0`.

[5] En 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 mon_backtrace

```
int mon_backtrace(int argc, char **argv, struct Trapframe *tf);
```

Parámetros:

- No se usan.

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

Descripción: El siguiente pseudocódigo describe el funcionamiento de `mon_backtrace`:

```
1  ebp = valor actual del registro ebp
2
3  eip = M[ebp + 1] // el eip (la dirección de retorno de la función) se halla
       una posición más atrás en la pila
4  arg0 = M[ebp + 2]
5  arg1 = M[ebp + 3]
6  arg2 = M[ebp + 4]
7  arg3 = M[ebp + 5]
8  arg4 = M[ebp + 6] // los parámetros se hallan detrás del eip en la pila
9
10 imprimir el ebp, el eip y los parámetros
11
12 si ebp == bootstacktop - 8:
13     retornar
14
15 ebp = M[ebp] // pasar al stack frame anterior en la cadena de llamadas (en
       la posición de memoria ebp se encuentra el ebp de la función anterior)
```

[6] *Page Directory Entry*
[7] *Page Table Entry*

```
16
17  ir a 3.
```

El *backtrace* finaliza cuando se alcanza el *stack frame* correspondiente a la función `i386_init`, que tiene a `bootstacktop - 8` como valor en el `ebp`. Dicha función es la primera función escrita en C a la que llama el código del *kernel*.

2.2.2. 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`.

2.2.3. 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.

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

Descripción: Esta función vuelve a ubicar la página 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.

2.2.4. 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.

Luego, si la tabla de páginas está presente, simplemente direccionamos sobre la misma utilizando `PTX(va)`, y guardamos un puntero a la entrada (PTE) obtenida en la posición apuntada por `pte_store`.

Si la tabla no está presente, devolvemos un error; a menos que el parametro `create` sea igual a 1. En tal caso, procedemos a reservar memoria física para la misma.

Para ello, intentamos obtener una página libre llamando a la función `page_alloc`; si esta no posee más memoria disponible, retornamos error. Si la llamada tiene éxito, obtenemos la dirección física de la página mediante `page2pa`, y escribimos en la PDE obtenida anteriormente la direccion física correspondiente, y ponemos los campos de presencia (`P`) y de escritura (`W`) como activos.

Finalmente, usando esta dirección física, calculamos su dirección virtual mediante `KADDR`, y le sumamos el valor de `PTX(va)` para obtener la dirección virtual de la PDE; valor que luego se guarda en el puntero al que referencia `pte_store`.

Por último, ponemos en 0 todos los bytes de la tabla, mediante `memset`, indicando que ninguna de las páginas de la misma se encuentra presente en memoria física.

2.2.5. Función `page_insert`

```
int page_insert(pde_t *pgdir, struct Page *pp, void *va, int perm);
```

[8] Estas 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.

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`.
- `pp` es un puntero a una página, la cual será asignada a la dirección virtual `va`.
- `va` es la dirección virtual/lineal a la cual se le asignará la página apuntada por `pp`.
- `perm` se utilizará para asignarle ciertos permisos a la PTE que apunte a la página.

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

Descripción: `page_insert` mapea una dirección virtual a una página física de la memoria principal. El procedimiente que se sigue es el descripto a continuación.

Primero se obtiene el PTE correspondiente a la dirección virtual especificada en `va` mediante una llamada a `pgdir_walk`. Si esta no finaliza satisfactoriamente, la función retorna con un error.

A continuación, se revisa si la PTE, para esa dirección `va`, se encuentra ya configurada para apuntar a una página física presente. Si esto es así, se revisa que la página que estaba presente no sea la misma que la que se está agregando — esto se hace comparando la dirección física de la página que se quiere mapear (resultado de `pagetopa()`), con la que estaba presente — y, en caso de ser una diferente, se la libera invocando a `page_remove`.

Luego, para todos los casos donde la página que se agrega no era la misma que estaba ya presente, se incrementa el contador de referencias de la misma en 1.

Finalmente, se escribe en la PTE correspondiente a esa dirección virtual, la dirección física de la página nueva, y se activan los bits de presente y los especificados en el parámetro `perm`.

2.2.6. Función `page_lookup`

```
struct Page * page_lookup(pde_t *pgdir, void *va, 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 correspondiente a la página que quiere ser obtenida.
- `pte_store` es un puntero a un puntero a una PTE. En el puntero al que hace referencia, se guardará la dirección de la PTE correspondiente a la dirección virtual/lineal `va`, si el valor de `pte_store` al llamar a la función era distinto de cero.

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

Descripción: Esta función obtiene la estructura `Page` correspondiente a la página física mapeada con la dirección virtual especificada en `va`. En el caso de que el parámetro `pte_store` sea distinto de 0 (cuando se invoca la función desde `page_remove`), se guarda, en el puntero al que referencia este mismo parámetro, la dirección de la PTE donde estaba mapeada la página.

Para obtener la página, primero se llama a `pgdir_walk` para obtener la PTE correspondiente a la dirección. Luego, a partir de esta PTE, se obtiene la dirección física de la página y, mediante una llamada a `pa2page`, se obtiene la estructura de la misma a partir de su direccion fisica. La dirección de esta estructura es la devuelta por la función.

Cuando el valor de `pte_store` es 0, se le asigna a esta variable la dirección de una variable local para poder llamar a `pgdir_walk` sin que esta genere errores.

En el caso de que la funcion `pgdir_walk` retorne un código de error, `page_lookup` devuelve 0.

2.2.7. Función `page_remove`

```
void page_remove(pde_t *pgdir, void *va);
```

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 correspondiente a la página que será liberada.

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

Descripción: Esta función llama a `page_lookup` para obtener la estructura de la página y la PTE correspondientes a la dirección virtual especificada en `va`.

Luego, revisa la PTE asociada a la misma, y, si el bit presente se encuentra activo en ella, establece el valor de la PTE obtenida en 0 (indicando que esa dirección virtual ya no posee soporte físico) y llama a `page_decref`, la cual descuenta en uno el valor del contador de referencias de la página y, si este alcanza el valor cero, vuelve a colocar la página en la lista de páginas libres. Por último, invalida el TLB (*Translation Lookaside Buffer*), debido a que ya no hay ninguna página fisica que corresponda a la dirección `va`.

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